Algorithms for planar graphs

by

Jittat Fakcharoenphol

B.E. (Kasetsart University) 1997
M.S. (University of California, Berkeley) 2001

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Satish Rao, Chair
Professor Alistair Sinclair
Professor Dorit Hochbaum

Spring 2003

The dissertation of Jittat Fakcharoenphol is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 2003

Algorithms for planar graphs

Abstract

Algorithms for planar graphs

by

Jittat Fakcharoenphol
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Satish Rao, Chair

This dissertation studies algorithms for cut and flow problems in planar graphs. Four results are presented.

In the first result, we give a faster algorithm for finding single-source shortest paths in planar graphs with negative weight edges. This problem has many applications. We show that one can solve such a problem in $O(n \log^3 n)$ time, improving on the results by Lipton, Rose, and Tarjan who gave an $O(n^{3/2})$ algorithm and by Henzinger, Klein, Rao, and Subramanian who gave a non-strongly polynomial time algorithm which runs in time $\tilde{O}(n^{4/3})$. We get this improvement by exploiting a non-crossing property of shortest paths in planar graphs. We also give a data structure that supports distance queries and updates.

The second result discusses multicommodity flows in planar graphs. We focus on a special case where every source lies on one face. This is a generalization of the Okamura-Seymour theorem. We show that in this case, the sparsest cut can be found in polynomial time. Furthermore, we show that if the cut condition is satisfied, one can find a set of paths that connects every source-sink pair, such that each edge in the graph is used no more than a constant number of times. This shows that the gap between the minimum cut and the integral flows is constant in this special case.

For the third result, we present a simple deterministic $O(n \log^2 n)$-time divide-and-conquer algorithm for finding minimum cuts in planar graphs. This can be compared to a randomized algorithm for general graphs by Karger that runs in time $O(m \log^3 n)$ and also the deterministic algorithm for general graphs by Nagamochi and Ibaraki that runs in time $O(mn + n^2 \log n)$. We use shortest paths in the dual graphs to partition the problem, and

use the relationship between minimum cuts in primal graphs and shortest paths in dual graphs to find minimum cuts that cross the partitions efficiently.

The last result in this dissertation is a graph decomposition procedure for graphs which exclude any fixed graph as a minor. We improve a result by Klein, Plotkin, and Rao, who showed that given an m-edge graph which excludes $K_{r,r}$ as a minor, one can find a decomposition such that at most $O(mr/\delta)$ edges are cut and each connected component has weak diameter at most $O(r^2\delta)$ for any parameter δ. This result has been used to get various approximation algorithms for such graphs. Their approach was to find good cuts from breadth-first-search trees starting from arbitrary nodes. We show that by choosing the starting nodes carefully one can get the same kind of decomposition where each component has weak diameter at most $O(r\delta)$.

Professor Satish Rao
Dissertation Committee Chair

Contents

Acknowledgments

The first thing you do before you start writing a thesis is thinking about people you want to thank. You plan everything; you have the first few drafts of it in your head. However, when everything boils down to actually writing it, you are almost speechless.

It is quite impossible to express how grateful I am in having Satish Rao as my advisor. I thank him for his encouragement both when I am in front of the white board and when I am behind the wheel, for his guidance, his patience, and for his support through out my precious years at Berkeley. Working closely with him has greatly shaped my approach to research.

I would like to thank my dissertation committee Alistair Sinclair and Dorit Hochbaum for their encouragement and their patience. I also thank Prof. Richard Karp for being in the qualifying examination committee. I thank the faculty members in the Theory group for making such a wonderful environment here at Berkeley.

I am grateful to the Thailand-U.S Educational Foundation (Fulbright); the Faculty of Engineering, Kasetsart University, Thailand; the Institute for the Promotion of Teaching Science and Technology, Thailand; and the National Science Foundation for financial supports.

I am indebted to a lot of friends. I thank my friends at the department, especially Dror Weitz, Steve Chien, Ziv Bar-Yossef, Chris Harrelson, Kunal Talwar, Kris Hildrum, Anupam Gupta, Sanjiv Das, Eduard Servan-Schreiber, and the mysterious group of people who were in The Poster Conspiracy; the Thai student community at Berkeley, especially Kriengsak Saokeaw, Chatchai Techayuenyong, Pitch Pongsawat, Vorapat Inkarojrit, Rachata Muneepeerakul, Pongpat Jirangpitakkul, and Tanapat Nakapanant; many of my friends from Thailand (I am sorry that I cannot list them here, but I am sure they all know); the girl I met at the photocopy machine who has inspired me over the years; and finally my roommates and ex-roommates: Chalermwut Chatdokmaiprai, Kittisak Yokthongwattana, Pipat Luengnaruemitchai, Tharinporn Iamsanitamorn, and Wanjira Jirajaruporn.

I would like thank my landlords for their hospitality. *Mama* instant noodles, "Lobo" magic cooking ingredients, and *Maesri* paste made me a bit more like home. The Thai Noodle Restaurant played a good part in it too. Thanks a lot to the Little Mandarin Restaurant for providing decent food for only $2.50.

I would like to greatly thank the people who gave me this precious opportunity of being

in Berkeley.

Finally, I would like to thank my family for their great support for my entire life.

Chapter 1

Introduction

Planar graphs arise quite naturally in various applications, for example, road maps, graphs created from images, finite element meshes, and VLSI layouts. This perhaps explains the long history of the study of planar graphs by graph theorists [Kur30, AH77a, AH77b, OS81] as well as more recently by theoretical computer scientists [HT74, LT79, LRT79, LT80, Fre89, WW95, KT95, MN95, HKRS97].

In this dissertation, we study path and cut problems in planar graphs. We present fast algorithms for finding shortest paths and minimum cuts, show the relationships between cuts and flows in some special case of the multicommodity flow problem, and finally discuss graph decompositions of metric spaces described by planar graphs.

1.1 Summary of the results

1.1.1 Shortest paths

We present an $O(n \log^3 n)$-time algorithm for finding shortest paths starting from a specific node in the graph with the negative weight edges. This improves over the algorithm by Lipton, Rose, and Tarjan [LRT79] which runs in time $O(n^{3/2})$ and the (non-strongly) polynomial-time algorithm by Henzinger, Klein, Rao, and Subramanian which runs in time $\tilde{O}(n^{4/3})$.

The algorithm that finds the shortest distance in a graph with negative weight edges must be able to determine whether there exists a cycle of negative length. In planar graphs, a cycle corresponds to a cut in the dual graph; therefore, the algorithm can be used to solve

many cut problems; for example, it is used as a subroutine in the Miller-Naor algorithm for testing flow feasibility.

We also present algorithms for query and dynamic versions of the shortest path problems.

The algorithms are discussed in Chapter 3. Lying at the heart of the algorithms are the data structures for the Monge searching problems which we describe in Chapter 4. This part of the dissertation is based on joint work with Satish Rao [FR01].

1.1.2 Edge-disjoint paths and multicommodity flows

Various problems in computer networks, e.g., path selection, routing, and network design, can be modeled using a capacitated graph with a set of source-sink pairs, or commodities. Given a graph with a set of source-sink pairs, the basic problem is to find a set of paths which connects all the source-sink pairs such that the paths mutually share no edges. It is not clear whether it is possible to do so. One of the natural necessary conditions that we investigate in this dissertation is the *cut criterion*, which states that the capacity, i.e., the number of edges, of every cut must be at least the number of commodities whose sources and sinks are separated by the cut. This condition is not sufficient in general. However, in many cases, it is. Okamura and Seymour [OS81] showed that in a planar graph where every source and sink lies on the boundary face the cut criteria is sufficient provided that some degree constraint is satisfied.

The cut condition is not sufficient when some source lies outside this particular face, as an example in Figure 1.1 shows. However, we show that if the cut criterion holds, one can find a set of paths that uses each edge at most a constant number of times. Specifically, for an instance of the problem in planar graph such that all sinks lie on the same face, we give an algorithm that finds a set of paths connecting all the source-sink pairs such that no edges are used more than 5 times, if the cut criterion holds.

We also give an algorithm that finds, in this special case, the worst cut, i.e., the cut that minimizes the ratio between the capacity of the cut and the demands across it.

We present this result, based on joint work with Satish Rao, in Chapter 5.

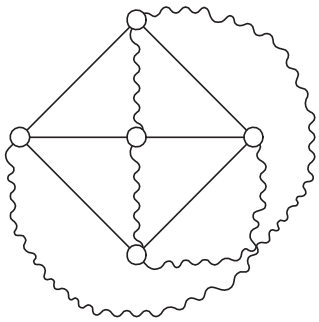


Figure 1.1: An instance such that the cut criteria holds, but the set of disjoint paths does not exist. The edges in the graph are shown in normal lines, while the source-sink pairs are shown in curly lines.

1.1.3 Minimum cuts

The *minimum cut problem* is to find the cut that minimizes the number of edges whose ends are separated by the cut. The size of the minimum cut is the weighted *edge-connectivity* of the graph. This problem is a basic problem in network design.

We give a simple *deterministic* algorithm for finding minimum cuts in planar graphs that runs in time $O(n \log^2 n)$. This can be compared to an $O(m \log^3 n)$-time randomized algorithm for general graphs by Karger [Kar00] and the best deterministic one by Nagamochi and Ibaraki [NI92] which runs in time $O(mn + n^2 \log n)$, which works for general graphs as well.

The algorithm is presented in Chapter 6. This is joint work with Parinya Chalermsook, Bundit Lekhanukit, Danupon Nanongkai, and Warat Yingsaeree.

1.1.4 Graph decomposition

Decomposing graphs into pieces with small diameter while, at the same time, cutting only a few edges is useful in various situations ranging from approximation algorithms [CKR01], dynamic object location algorithms [AP90], to the area of metric embeddings [Rao99]. Klein, Plotkin, and Rao [KPR93] showed that this can be done in the case of graphs which exclude a fix graph as a minor. Their algorithm has been used as a subroutine in approximation algorithms for various problems in graphs which exclude some fixed minor, e.g., planar graphs.

For any parameter δ and r, Klein, Plotkin, and Rao gave a procedure that given an

n-node m-edge graph, either finds a $K_{r,r}$ minor or a set of $O(mr/\delta)$ edges whose removal leaves components of weak diameter $O(r^2\delta)$. We improve the dependence on r in the weak diameter of the components. Specifically, we give a decomposition procedure which either finds a K_r minor or a decomposition that cuts at most $O(mr/\delta)$ edges such that each component has weak diameter $O(r\delta)$. Note that we find, instead of $K_{r,r}$, a complete graph minor K_r. This result, however, generalizes to the case of graphs which exclude $K_{r,r}$ as a minor, since $K_{r,r}$ is a minor of K_{2r}.

This result which is joint work with Kunal Talwar [FT03] is presented in Chapter 7.

Chapter 2

Preliminaries

In this chapter we discuss important properties of planar graphs that we use later on. In section 2.2, planar graph duality is presented along with its applications. We mention the separator theorem with its extension in section 2.3. The facts presented here are not meant to be comprehensive. For more information, see, for example, [Bol98, Die00, Har71].

2.1 Basic facts

Planar graphs are graphs which can be drawn on the plane without any crossing edges. One combinatorial characterization of planar graphs is by the theorem of Kuratowski [Kur30]. Graph G contains graph H as a minor if one can obtain H from G by a sequence of edge deletions and contractions. Kuratowski's theorem states that planar graphs are graphs that do not contain K_5 or $K_{3,3}$ as a minor. In general, for any fixed graph H, determining whether a given graph contains H as a minor can be done in cubic time[1] [RS95]. For planar graph, a linear-time algorithm for planarity testing was given by Hopcroft and Tarjan [HT74].

A plane embedding of a planar graph divides the plane into regions called faces. Planar graphs are sparse graphs. Given a planar graph G, Euler's Polyhedron Formula states that

$$n - m + f = 2,$$

where n is the number of nodes, m is the number of edges, and f is the number of faces. If the graph is simple, i.e., there exists at most one edge for each pair of nodes, one can

[1] The constant in the running time, however, depends severely on the size of H.

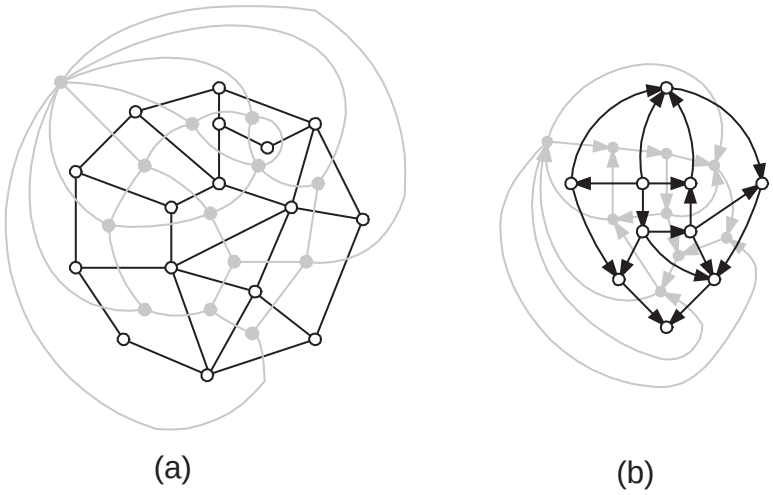


Figure 2.1: Planar graphs and their dual graphs (shown in gray): (a) undirected graph and (b) directed graph.

further show that $m \leq 3n - 6$.

2.2 Duality

When a planar graph is drawn on a plane, one can exploit its topological structure. Given any undirected planar graph G with an embedding on a plane, a *dual undirected graph* G^* can be constructed as follows. The nodes of G^* correspond to faces in G. If, in G, face f_1 shares some edge e with face f_2, we have a corresponding dual edge $e^* = (f_1^*, f_2^*)$ in G^* where f_1^* and f_2^* are nodes in the dual graph corresponding to f_1 and f_2, respectively. We also call G the *primal* graph. We note that the dual of G^* is G itself. Figure 2.1(a) gives an example of a planar undirected graph and its dual.

A dual graph for directed planar graph can be defined in the same manner. Intuitively, a dual edge goes across the corresponding edge in the primal graph. In the case of directed graph, we direct a dual edge to go to, say, the right side of its primal edge. See Figure 2.1(b) for example.

The correspondence between primal edge e and dual edge e^* above extends to the correspondence between cuts and cycles. We claim without proof that a cut in G corresponds to a cycle in G^*. This is a very useful topological property that is used over and over again,

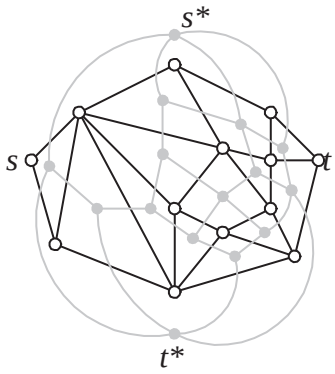


Figure 2.2: Finding an st-minimum cut.

because, in many cases, dealing with cycles is easier than with cuts[2]. In fact, a few NP-hard problems in general graphs can be solved on planar graphs thanks to this duality; our favorite examples include the feedback arc set problem [LY78] and the minimum-quotient cut problem [PP93].

We are not constrained to use the dual exactly as it is. Sometimes, we modify it to suit our need. To illustrate, we show the algorithm by Hassin [Has81] for finding minimum st-cut for the case that s and t lie on the same face. For any planar graph with an embedding on a plane, one can turn any face to be a boundary face. Therefore, we can assume that s and t lie on the boundary face. The edges on the boundary face can be divided into 2 paths, the top path and the bottom path as in Figure 2.2. We create the dual graph as discussed previously, with the exception that we now have 2 dual nodes s^* and t^* for the boundary face. The dual edges corresponding to the edges on the top path go to s^*, and the rest goes to t^*. We can see the correspondence between an st-cut and a path from s^* to t^*; hence, to find the minimum st-cut, we need to find the shortest path from s^* to t^* in the dual graph, which can be done in linear time using the algorithm by Henzinger, Klein, Rao, and Subramanian [HKRS97].

[2]We also note that we have far more technologies for dealing with paths and cycles than for dealing directly with cuts; apparently we often return to work with flows, which are indeed sets of paths, in order to get a handle on cuts.

2.3 Small separators

The well-known graph decomposition technique for planar graphs is based on the separator theorem by Lipton and Tarjan [LT79]. We say that an n-node graph G has an $f(n)$-*node separator* if there exists a partition of the nodes into three sets A, B, and C such that $|C| \leq f(n)$, the size of A and B are at most $2n/3$, and no edge exists between A and B. The theorem of Lipton and Tarjan states that given an n-node planar graph, one can find, in linear time, a node separator of size $2\sqrt{2 \cdot n}$, which was later improved by Djidjev [Dji82] to $\sqrt{6}\sqrt{n}$.

The separator theorem has been used in many divide-and-conquer algorithms, for examples, algorithms for finding VLSI layouts for planar graphs [Lei80, Val81], the nested dissection algorithm in numerical analysis [LRT79], and polynomial time approximation schemes for various NP-optimization problems over planar graphs [LT80]. The general framework of divide-and-conquer works as follows. Given a graph, the algorithm works by finding the node separator which is used to partition the graph into two subgraphs, then the algorithm recursively finds separators in these subgraphs; this produces a *separator decomposition*. By using Lipton-Tarjan's algorithm recursively one can find such a decomposition in time $O(n \log n)$. Goodrich [Goo95] showed how to construct a separator decomposition of the graph in linear time.

The separator algorithms discussed above use a breadth-first search tree to decompose the graph using a series of cuts based on the levels of edges in the tree. Therefore, the pieces in the decomposition might not be "nice" topologically. In our algorithm we use Miller's cycle separator algorithm [Mil86]. Given a 2-connected triangulated planar graph with n nodes, Miller's algorithm finds a simple cycle separator of length at most $2\sqrt{2n}$ in linear time.

Chapter 3

Shortest path algorithms

3.1 Overview

The shortest path problem with real (positive and negative) weights is the problem of finding the shortest distances from a specified source node to all the nodes in the graph. For this paper, we assume that the graph actually has no negative cycles since the shortest path between two nodes will be undefined in the presence of negative cycles.

In general, algorithms for the shortest path problem can, however, easily be modified to output a negative cycle, if one exists. This also holds for the algorithms we present here.

The shortest path problem has long been studied and continues to find applications in diverse areas. The problem has wide application even when the underlying graph is a grid. For example, there are recent image segmentation approaches that use negative cycle detection [CRZ96, DGV95]. Some of our other favorite applications for planar graphs include separator algorithms [Rao92], multi-source multi-sink flow algorithms [MN95], or algorithms for finding minimum weighted cuts.

In 1958, Bellman and Ford [Bel58, FF62] gave an $O(mn)$ algorithm for finding shortest paths on an m-edge, n-vertex graph with real edge weights. Gabow and Tarjan [GT89] showed that this problem could indeed be solved in $\widetilde{O}(m\sqrt{n})$.[1] Their algorithm, however, depended on the values of the edge weights. For strongly polynomial algorithms, Bellman-Ford remains the best known.

As for graphs with positive edge weights, the problem is much easier. For example, Dijkstra's shortest path algorithm [Dij59] can be implemented in $O(m + n \log n)$ time.

[1] The $\widetilde{O}(\cdot)$ notation ignores logarithmic factors.

For planar graphs, upon the discovery of planar separator theorems [LT79], an $O(n^{3/2})$ algorithm was given by Lipton, Rose, and Tarjan. [LRT79]. Their algorithm is based on partitioning the graph into pieces and recursively computing distances on the borders of the pieces using numerous invocations of Dijkstra's algorithm to build a dense graph. Then they use the Bellman-Ford algorithm on the resulting dense graph to construct a global solution. Their algorithms worked not only for planar graphs but for any $\sqrt{n}$-separable one.[2]

Combining a similar approach with a (non-strongly) polynomial algorithm of Goldberg [Gol92] for general graphs, Henzinger, Klein, Rao, and Subramanian [HKRS97] gave an $\widetilde{O}(n^{4/3})$ algorithm for the shortest path problem on planar graphs (or any set of graphs with an $O(\sqrt{n})$ sized separator).

In this chapter, we present an $O(n \log^3 n)$ time algorithm for finding shortest paths in a planar graph with real weights.

We also present algorithms for query and dynamic versions of the shortest path problems.

3.1.1 The idea

Our approach is similar to the approaches discussed above in that it constructs a rather dense non-planar graph on a subset of nodes and then computes a shortest path tree in that graph.

We observe that there exists a shortest path tree in this dense graph that must obey a non-crossing property in the geometric embedding of the graph inherited from the embedding of the original planar graph. Using this non-crossing condition, we can compute a shortest path tree of the dense graph in time that is nearly linear in the number of nodes in the dense graph and significantly less than linear in the number of edges. Specifically, we decompose our dense graph into a set of bipartite graphs whose distance matrices obey a non-crossing condition (called the Monge condition). Efficient algorithms for searching for minima in Monge arrays have been developed previously. See, for example, [ABNK$^+$95, BY98].

Our algorithm proceeds by combining Dijkstra's and the Bellman-Ford algorithms with methods for searching Monge matrices in sublinear time. We use an on-line method for

[2] Assuming that they were given a recursive decomposition of the graph.

searching Monge arrays with our version of Dijkstra's algorithm on the dense graph.

We note that our algorithms rely heavily on planarity, whereas some of the previous methods only require that the graphs are separable.

3.1.2 Our results

We give the following results.

- An $O(n \log^3 n)$ algorithm for finding shortest paths in planar graphs with real weights.
- An algorithm that requires $O(n \log^3 n)$ preprocessing time and answers distance queries between pairs of nodes in time $O(\sqrt{n} \log^2 n)$.

  The best previous algorithms had an $\Theta(n^2)$ query-preprocessing time product, whereas ours is $\widetilde{O}(n^{3/2})$.
- An algorithm that supports distance queries and update operations that change edge weights in amortized $O(n^{2/3} \log^{7/3} n)$ time per operation. This algorithm works for positive edge weights.
- An algorithm that supports distance queries and update operations that change edge weights in amortized $O(n^{4/5} \log^{13/5} n)$ time per operation. This algorithm works for negative edge weights as well.

We also present an on-line Monge searching problem and methods to solve it that may be novel and of independent interest.

3.1.3 More related work

For planar graphs with positive edge weights, Henzinger et al. [HKRS97] gave an $O(n)$ time algorithm. Their work improves on work of Frederickson [Fre89] who had previously given $O(n\sqrt{\log n})$ algorithms for this problem.

Frederickson [Fre87] gave an improved all-pairs shortest path algorithm for planar graphs with small hammock decompositions. Djidjev et al. [DPZ91] gave dynamic algorithms whose complexity are linear in the size of the hammock decomposition. This could be quite efficient in certain cases, e.g. when the graph is outerplanar. But for general planar graphs–even grid graphs–their algorithms are no better than those in [LRT79].

For approximate distance queries in planar graphs with non-negative weight edges, Thorup [Tho01] showed that one can construct a nearly linear-size data structure in near-linear time which can answer a $(1+\varepsilon)$ approximate distance in $O(\log\log\Delta + 1/\varepsilon)$ time where Δ is the longest finite distance in the graph.

In unweighted planar graphs, Kowalik and Kurowski [KK03] show how to answer each query in constant time for paths of constant length.

A binary searching technique similar to the one we use in the Monge searching problem also appeared in an algorithm for finding shortest paths on a three-dimensional polygon by Mitchell et al. [MMP87].

3.1.4 Organization of the chapter

We review the basic algorithms and techniques in section 3.2. In section 3.3, we define our main tool, the dense distance graph, which is an efficiently searchable representation of distances in the planar graph. Section 3.4 describes how to compute the graph inductively, by relying on some Monge data structures and efficient implementations of Dijkstra's algorithm and the Bellman-Ford algorithm. We show how to use it to compute a shortest path labelling of the graph in section 3.5. Sections 3.6.1 to 3.6.3 show how to use the dense distance graph as the basis for query and dynamic shortest path algorithms.

3.2 Preliminaries

Given a directed graph $G = (V, E)$, and a weight function $d : E \to R$ on the directed edges, a distance labelling for a source node s is a function $d : V \to R$ such that $d(v)$ is the minimum over all s to v paths P of $\sum_{e\in P} d(e)$.

3.2.1 Algorithms

In this subsection, we describe two algorithms that we use: Dijkstra's algorithm and the Bellman-Ford algorithm. Both algorithms work through a sequence of edge relaxations. They start with a labelling $d(\cdot)$ and choose an edge to relax. The relax operation proceeds for an edge $e = (u, v)$ by setting the distance label $d(v)$ to the minimum of $d(v)$ and $d(u) + d(e)$.

3.2.1.1 Dijkstra's algorithm

Dijkstra's algorithm (described below) correctly computes a distance labelling when the weights on the edges are nonnegative, i.e., $d(e) \geq 0$ for all $e \in E$.

Algorithm Dijkstra

1. $d(s) = 0, d(v) = \infty, \forall v \neq s$.
2. $S = \{s\}$.
3. while $S \neq V$ do

3.1 $u = \text{findMin}_d(V \setminus S)$

3.2 foreach $e = (u, v)$

3.2.1 $d(v) = min(d(v), d(u) + d(e))$ /* This is an edge relaxation */

3.3 $S = S \cup \{u\}$

The algorithm maintains the set S of *scanned* nodes. For each iteration, the unscanned node u with the smallest distance label is chosen and scanned. We then add u into S. The algorithm uses $O(m)$ edge relaxations and another $O(n)$ findMin operations which returns the node in $V \setminus S$ with the smallest distance label.

3.2.1.2 The Bellman-Ford algorithm

For distance functions where $d(e)$ could be less than zero, Bellman and Ford suggested the following algorithm, which is guaranteed to compute a distance labelling if there is no cycle in the graph whose total weight under $d(\cdot)$ is negative.

Algorithm Bellman-Ford

1. $d(s) = 0, d(v) = \infty, \forall v \neq s$.
2. $phase = 0$.
3. while $phase \leq n$ do

3.1 relax all edges.

3.2 $phase \leftarrow phase + 1$.

After phase i, all nodes which are i hops away from s get the correct distance label. Since all nodes are within n hops, after n phases, all nodes get the right distance label. In many case, including ours, we have an upperbound on the number of hops; thus, we can run the algorithm with less than n phases.

3.2.2 Feasible price functions and relabellings

Let a function $p : V \to R$ be a *price function* over the node set. The *reduced cost function* d_p over the edge set induced by the price function p is defined as

$$d_p(e = (u, v)) = p(u) + d(e) - p(v).$$

It is well-known that the reduced cost function preserves the presence of negative cycles and also the shortest paths [Joh77]. Consider any path P from s to t. The original distance of P is $\sum_{e \in P} d(e)$, while the distance under the reduced cost is $\sum_{e \in P} d_p(e) = p(s) - p(t) + \sum_{e \in P} d(e)$. Note that the difference does not depend on the path; therefore, the shortest path remains the same. Furthermore, in the case of a cycle, where $s = t$, the distance remains the same.

We say that the price function p is *feasible* if and only if for all edges $e = (u, v)$, $d_p(u, v) \geq 0$. Hence, for any feasible price function p, we can find a distance labelling from any source node using Dijkstra's algorithm on the modified graph with d_p as weights (called the *relabelled graph*). The distance labelling for the original graph can be easily recovered from the relabelled one. We note that a valid set of distance labels for any source node is a feasible price function.

Thus, computing shortest paths from k sources in a graph with negative weight edges can be accomplished with only one application of the Bellman-Ford algorithm and $k - 1$ applications of Dijkstra's algorithm.

3.3 The dense distance graph

A *decomposition* of a graph is a set of subsets $S_1, S_2, \ldots, S_k$ (not necessarily disjoint) such that the union of all the sets is V and for all $e = (u, v) \in E$, $u, v \in S_i$ for some i. A node v is a border node of a set S_i if $v \in S_i$ and there exists an edge $e = (v, x)$ where $x \notin S_i$. We refer to the subgraph induced on a subset S_i as a piece of the decomposition.

We assume that we are given a recursive decomposition where at each level, a piece with n nodes and r border nodes is divided into two subpieces such that each subpiece has no more than $2n/3$ nodes and at most $c\sqrt{n}$ border nodes. (The recursion stops when a piece contains a single edge.)

In this recursive context, we define a border node of a subpiece to be any border node of the original piece or any new border node introduced by the decomposition of the current

piece.

It is convenient to define the level of a decomposition in the natural way, with the entire graph being the only piece in the level 0 decomposition, the pieces of the decomposition of the entire graph being the level 1 pieces in the decomposition, and so on. A node is a level i border node if it is a border node of a level i piece. Note that a node may be a border node for many levels. Indeed, any level i border node is also a level j border node for all $j > i$.

A *hole* is a bounded face all nodes adjacent to which are border nodes. For simplicity, we assume inductively that there is a planar embedding of any piece in the recursive decomposition where all the border nodes are on a single face and are circularly ordered. This assumption implies that there is no hole in every piece in the decomposition. Although this assumption is true for interesting classes of planar graphs, e.g, grid graphs, it is not true in general. In section 3.7, we show how to generalize the algorithm to work with a piece with a constant number of holes and describe how one can find a recursive decomposition of that form in $O(n \log n)$ time.

We assume, without loss of generality, that the graph is a bounded-degree graph and, for simplicity, that each piece is connected.

For each piece of the decomposition, we recursively compute the all-pairs shortest path distances between all its border nodes along paths that lie entirely inside the piece.

We call this the *dense distance graph* of the planar graph. The *level i dense distance graph* is the subgraph of the dense distance graph on the level i border nodes. We refer to the *level i dense distance graph of a piece* as the subgraph of the level i dense distance graph whose edges correspond to paths that lie in the piece.

This graph underlies previous algorithms for shortest paths in planar graphs. We give a better algorithm to construct and use it.

3.4 Computing the dense distance graph

We assume (recursively) while computing the level i dense distance graph that we have the level $i + 1$ graph and the distances between all the border nodes of each piece. We also assume that the piece contains n nodes and $c\sqrt{n}$ border nodes. By a property of the decomposition, we assume that each of the two subpieces of P contain at most $c'\sqrt{n}$ border nodes. Thus, the level $i + 1$ dense distance graph contains at most $O(\sqrt{n})$ nodes.

We will show how to find the edges of the level i dense distance graph that correspond to a particular piece P. Our algorithm follows the same idea as the Lipton-Rose-Tarjan algorithm, which we present briefly in section 3.4.1.

Their algorithms, however, used implementations for the Bellman-Ford and Dijkstra's algorithms which depended linearly on the number of edges that are present in the level $i+1$ dense distance graph. Our methods depend near linearly on the number of nodes in the dense distance graph, which is the square root of the number of edges. Our implementation of the Bellman-Ford is presented in section 3.4.2. Section 3.4.3 describes the implementation of the Dijkstra's algorithm.

3.4.1 Lipton-Rose-Tarjan $O(n^{3/2})$ algorithm

Recall that the level i dense distance graph for P consists of the all-pairs shortest path distances between border nodes of its subpieces in the level $i+1$ dense distance graph.

Also, note that the level $i+1$ dense distance graph may contain negative edges. By finding a feasible price function using a single Bellman-Ford computation from any source, however, we can find the shortest path distances from any other source using only the Dijkstra computation as stated in section 3.2.2.

We proceed by doing a single Bellman-Ford computation in the level $i+1$ dense distance graph of P from one border node, and then doing $r-1$ Dijkstra computations on the relabelled graph to compute the shortest path distances from the remaining border nodes. (Recall that r is the number of border nodes.)

There are $O(\sqrt{n} \cdot \sqrt{n}) = O(n)$ edges in the level $i+1$ dense distance graph of P, and there are $r = O(\sqrt{n})$ border nodes. Thus, the running time for the Bellman-Ford computation is $O(n \cdot \sqrt{n}) = O(n^{3/2})$. We run r computations of the Dijkstra's algorithm, which take $O(r \cdot (n + \sqrt{n} \log n)) = O(n^{3/2})$ time. This gives the running time of $O(n^{3/2})$ for computing the level i dense distance graph. The time to compute the top level dense distance graph dominates; thus, the time for computing the dense distance graph is $O(n^{3/2})$.

3.4.2 Implementing the Bellman-Ford step

The Bellman-Ford algorithm that we run proceeds as follows.

Algorithm Bellman-Ford

1. $d(s) = 0, d(v) = \infty, \forall v \neq s$.

2. $phase = 0$.
3. while $phase \leq O(\sqrt{n})$ do
3.1 relax all edges.
3.2 $phase \leftarrow phase + 1$.

The total number of boundary nodes in each subpiece of P is $O(\sqrt{n})$, so the number of edges is $O(n)$. Therefore, if we relax every edge directly as in [LRT79], the running time for each step of edge relaxation would be $O(n)$ for all of P. The total running time for the Bellman-Ford step would then be $O(n^{3/2})$.

However, we will relax the edges in time that is nearly linear in the number of nodes, in particular $O(\sqrt{n}\log^2 n)$ time. This gives a running time of $O(n\log^2 n)$ for the Bellman-Ford step.

We accomplish this by maintaining the edges of each subpiece of P in $O(\log n)$ levels of Monge arrays. The edges in each Monge array can be relaxed in $O(k \log k)$ time, where k is the number of nodes in the data structure.

The first Monge array that we define is formed as follows. Divide the border nodes in some subpiece into two halves, the first (or left) half in the circular order (with an arbitrary starting point) and the second (or right) half. Consider the set of edges in the dense distance graph that go from the left border nodes to the right ones. The edges obey the Monge property, since the underlying shortest path tree need not cross.

Using the same left-right partitioning, we can define another Monge array with the direction of edges reversed (i.e., edges in the array go from the right border nodes to the left border nodes).

Successive Monge arrays are constructed by recursively dividing the left and right halves further. Each node will occur in at most $O(\log n)$ data structures, and each edge will occur in one data structure. Figure 3.1 shows how we partition nodes and edges between them.

We can relax all the edges in a Monge array as follows. The nodes on the left have a label associated with them, and a node v on the right must choose a left node u which minimizes $d(u) + d(u, v)$. However, because of the planarity of the piece, the "parent" edges of two right nodes need not cross, and this gives us the Monge property. For this special case, we can use a standard divide-and-conquer technique to find all the parents in time $O(r \log r)$, where the number of nodes in the Monge array is r.

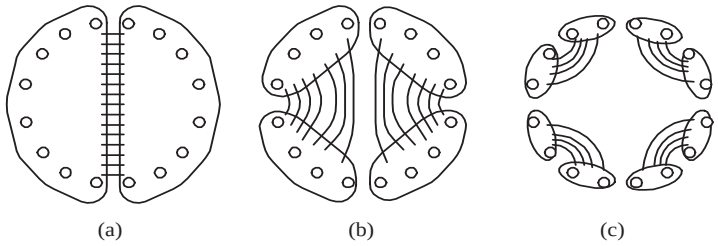


Figure 3.1: Partitions of nodes to form $O(\log n)$ Monge arrays: (a) the first and second arrays, (b) the third and forth arrays, and (c) the fifth and sixth arrays.

The total number of nodes in all the data structures is $O(\sqrt{n}\log n)$ for each subpiece of P in the decomposition.

In P we have two subpieces. Therefore, the time for relaxing all of P's edges is $O(\sqrt{n}\log^2 n)$. The number of phases the Bellman-Ford runs is the number of nodes in the longest path, which is $O(\sqrt{n})$. Thus, the time for the Bellman-Ford step is $O(n\log^2 n)$ on an n-node piece.

3.4.3 Implementing the Dijkstra step

After one invocation of Bellman-Ford, we have a shortest path tree from some border node of P. Now, using the relabelling property, we can modify all edge weights so that they are all positive and the shortest paths remain unchanged. With these modified weights, we repeatedly apply Dijkstra's algorithm to compute all-pairs shortest distances among the border nodes in P.

In order to compute the shortest path distances from each border node s of P, we proceed as in Dijkstra.

While working at level i of the decomposition, we view the subpieces at level $i+1$ each separately. Each subpiece maintains a data structure that allow us to scan a node (relax all edges in the dense distance graph adjacent to that node in that subpiece) and find the minimum labelled node in the subpiece efficiently.

As in Dijkstra's algorithm, we will maintain a set of scanned nodes S and a global heap H for keeping minimum labelled nodes from all subpieces. A node is extracted from the global heap. This node can belong to many subpieces, so we scan it in all the subpieces containing it. After we scanned the node, the minimum labelled node in some subpiece

Algorithm Dijkstra-Step

1. $d(s) = 0, d(v) = \infty, \forall v \neq s$.
2. $S \leftarrow \{s\}$.
3. for all $S_i \ni s$ do
4. AddToHeap$(H, (s, S_i))$.
5. while H is not empty do
5.1 $S_{min} \leftarrow$ ExtractMin(H).
5.2 $v \leftarrow$ ExtractMinInSubpiece(S_{min}).
5.3 if $v \notin S$,
5.3.1 for all $S_i \ni v$ do
5.3.1.1 ScanInSubpiece(S_i, v, d_v).
5.3.1.2 UpdateHeap$(H,$ FindMinInSubpiece$(S_i), S_i)$.
5.4 else
5.4.1 UpdateHeap$(H,$ FindMinInSubpiece$(S_{min}), S_{min})$.
5.5 $S \leftarrow S \cup \{v\}$.

Figure 3.2: Pseudocode for our Dijkstra implementation.

might change, so we have to update the entry of that subpiece in the global heap H.

The primary difference between our implementation of Dijkstra's algorithm and the normal one is that in our implementation a node that is already scanned can appear again in the heap. This is because the data structure in each subpiece does not guarantee that a minimum border node after being scanned will never reappear as a minimum node again. The data structure does, however, guarantee that that node can reappear at most $O(\log r)$ times.

Let S_i denote the subpieces in P. The pseudocode in figure 3.2 describes the algorithm for computing the shortest path tree starting at a border node s. It uses the following operations on data structures that are maintained for each of the subpieces.

- ScanInSubpiece(S_i, v, d_v): Relax all edges of v in the dense distance graph at level $i+1$ in piece S_i conditioned on $d(v) = d_v$.

  A sequence of l calls to ScanInSubpiece can be implemented in $O(l \log^2 r)$ time.

- FindMinInSubpiece(S_i): Return the border node (which might already be scanned) in piece S_i whose label is no greater than all unscanned nodes in the piece.

  This procedure can be implemented in $O(1)$ time.

- ExtractMinInSubpiece(S_i): Return the border node (which might already be scanned) in piece S_i whose label is no greater than all unscanned nodes in the piece, and attempt to remove it from the heap in the piece. We say 'attempt' because the extracted node can be returned as the minimum node again, however no node can be returned by this procedure more than $O(\log r)$ times.

  A sequence of l calls to ExtractMinInSubpiece can be implemented in $O(l \log r)$ time.

This data structure can be implemented directly using the on-line non-bipartite Monge searching data structure in section 4.3. It is constructed after the Bellman-Ford step in time linear in the number of edges, i.e., in time $O(r^2) = O(n)$. For each subpiece S_i, we keep the data structure; for ScanInSubpiece, we call Activate, for FindMinInSubpiece, we call FindMin, and for ExtractMinInSubpiece, we call ExtractMin. We defer the implementation details of the Monge searching data structure to the next chapter. At this point we assume the bounds stated above and use them to bound the running time of the Dijkstra step.

We stress that an already scanned node might be returned from FindMinInSubpiece and ExtractMinInSubpiece. The data structure does, however, guarantee that any border nodes will not be returned from ExtractMinInSubpiece more than $O(\log r)$ times.

When the data structure for each subpiece returns a minimum labelled unscanned node, it is the minimum unscanned node in the subpiece. Also, the algorithm only scans a node which is the minimum over all nodes returned from all the subpieces. Therefore, every time the algorithm scans a node v, v has the minimum distance label over all unscanned nodes.

Thus, our algorithm is a valid implementation of Dijkstra's algorithm in that it only scans the minimum labelled nodes. Thus, it correctly computes a shortest path labelling.

3.4.3.1 Analysis of the running time of the Dijkstra step

For this piece, there are $O(r) = O(\sqrt{n})$ border nodes in consideration.

The number of calls to ScanInSubpiece is bounded by the number of nodes in the level $i + 1$ dense distance graph. (Because the degree for each node is bounded, the node

belongs to a constant number of subpieces.) Since each node is scanned once as in Dijkstra's algorithm, there are $O(r)$ calls to ScanInSubpiece.

ExtractMinInSubpiece is called at most $O(\log r)$ times on each node in the level $i+1$ dense distance graph. So the total number of operations is $O(r \log r)$ for a total cost of $O(r \log^2 r)$.

Finally, the number of calls to FindMinInSubpiece is bounded by the number of calls to ScanInSubpiece for a subpiece and to ExtractMinInSubpiece.

The time for manipulating the heap H is $O(1)$ because there are a constant number of subpieces.

Therefore, the running time for computing each shortest path tree is $O(r \log^2 r) = O(\sqrt{n} \log^2 n)$, and the total running time for computing all trees is $O(\sqrt{n}\sqrt{n} \log^2 n) = O(n \log^2 n)$.

3.4.4 The running time for constructing the dense distance graph

For each level of the decomposition, the time for doing the Bellman-Ford step is $O(n \log^2 n)$, and the time for all Dijkstra computations is $O(n \log^2 n)$.

Since there are at most $O(\log n)$ levels in the decomposition, the time to construct the whole dense distance graph is $O(n \log^3 n)$.

3.5 Computing shortest path

To actually solve the shortest path problem for a source s, we use the dense distance graph as follows.

We add s as a "border" node to all the pieces that contain it and compute the dense distance graph on the resulting decomposition. We compute a shortest path labelling for the source s in the level 1 dense distance graph to the border nodes using the Bellman-Ford algorithm above.

We then extend the distances to the internal nodes recursively, again, using the Bellman-Ford algorithm.

The Bellman-Ford computation costs $O(n \log^2 n)$ for each level. Therefore, the running time for computing all the distances is $O(n \log^3 n)$.

3.6 Dynamic algorithms

3.6.1 Supporting queries when the graph is static

The dense distance graph and the Dijkstra procedure above can be used to answer shortest path queries between a pair of nodes. In this section we show how to use the dense distance graph to find the shortest distance between any pair of nodes in $O(\sqrt{n}\log^2 n)$ time.

The algorithm for this is very similar to the one for the Dijkstra step in the shortest path algorithm. Suppose the query is for the distance of a pair (u, v). The shortest (u, v) path can be viewed as a sequence of paths between border nodes of the nested pieces that contain u and v. The lengths of these paths is represented in the dense distance graph as an edge between border nodes and enclosing border nodes or as an edge among border nodes of a piece. Thus, we can perform a Dijkstra computation on this subgraph of the dense distance graph to compute the shortest (u, v) path.

We derive the bound on the number of border nodes in the pieces containing u as follows. Each piece, except the first one, which is G, is a piece in the decomposition of another piece. Hence the number of nodes goes down geometrically. Also, the number of border nodes, which is bounded above by the square root of the number of nodes in the piece, goes down geometrically. Therefore the number of border nodes involved is $O(\sqrt{n})$.

We use the same algorithm as in the Dijkstra step, but now we work with many pieces from many levels of the decomposition. The algorithm in the Dijkstra step continues to work in time $O(k \log^2 n)$ where k is the total number of nodes involved in the Dijkstra step.

Since the total number of nodes in the graph that we are searching is $O(\sqrt{n})$, the running time is bounded by $O(\sqrt{n}\log^2 n)$.

3.6.2 Dynamic algorithms for graphs with only positive edge weights

A dynamic data structure answers shortest path queries and allows edge cost updates, where the cost of an edge may be decreased or increased. (Edge additions and deletions are not addressed in this paper.)

The query algorithm in the previous subsection works only with the pieces in the recursive decompositions that contain the query pair. It can avoid the other pieces because all the distances in those pieces are reflected in the distances among the border nodes of the pieces containing them.

In the dynamic version, we will use the same algorithm as in the query-only case. Since our query step uses Dijkstra's algorithm, it is crucial that all weights are non-negative. However, some update might introduce a negative edge in the relabelled graph. To simplify the presentation, we first discuss the case that all edges have positive weights in this section. In the following section we extend the idea to the general case.

We do not know how to efficiently maintain an explicit representation of the dense distance graph when an update occurs. But, only the pieces containing an update edge will not have the correct distances among their border nodes. That is, any edge in the dense distance graph between two border nodes of a region containing an update edge no longer has an accurate distance label. Any other edge in the dense distance graph has the correct label.

We call the pieces that contain updated edges *activated pieces* and call the border nodes of these pieces *activated nodes*. (See figure 3.3(b) for example.)

To properly recompute the distance for a piece p that contains an update edge, we need to consider the distances among all the border nodes of pieces that are contained in p. Thus, we define the *activated graph* to be all the valid edges corresponding to border nodes of the pieces containing an update edge and their sibling pieces. (See figure 3.3(c).)

We answer a query for a pair (u, v) by adding the valid edges of border nodes of pieces containing u and v to the activated graph and running a Dijkstra's computation on the resulting graph. We call this graph the extended activated graph for (u, v). (See figure 3.3(d).)

We proceed by deriving a bound on the number of nodes involved in the computation assuming that we allow a maximum of k updates before rebuilding the entire data structure.

For each update, the number of border nodes on the pieces that need to be in Dijkstra's computation is $O(\sqrt{n})$. Naively, one can bound the total number of activated nodes by $O(k\sqrt{n})$. However, if we consider a top-down process that divides any piece that contains an update edge, we can show that the total number of activated nodes is $O(\sqrt{nk})$ as follows.

Consider the decomposition tree. There are at most k leafs that are activated. Hence, at most $k - 1$ pieces have both their children activated; call these pieces *branching* pieces. Because the number of nodes goes down geometrically along the tree, we can bound the total number of activated border nodes using the number of border nodes of the branching pieces. The worst case is that all $k - 1$ branching pieces are in the highest level of the decomposition tree, i.e., they form a balanced binary tree. We note that the pieces on the

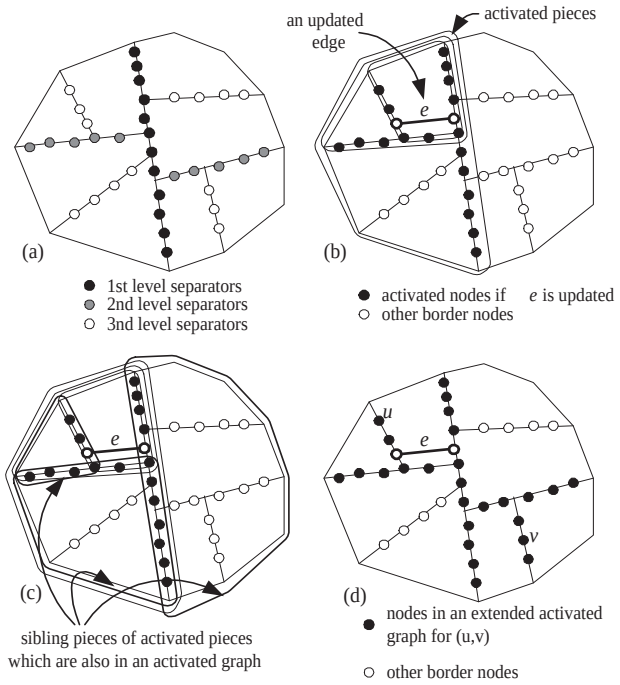


Figure 3.3: (a) A graph with a three-level decomposition. (b) Activated nodes for (u, v). (c) An activated graph when e is updated. (d) An extended activated graph for (u, v).

same level partition the the graph; thus, the number of border nodes is maximized when they partition the graph evenly. Hence, on level l there are at most $2^l\sqrt{n/2^l} = \sqrt{n2^l}$ border nodes. The sum on the last level dominates the total sum; therefore, the number of border nodes is $O(\sqrt{nk})$.

Thus, the Dijkstra computation described in section 3.4.3 will run in time $O(\sqrt{nk}\log^2 n)$.

Therefore, the total cost of a sequence of k updates and queries is $O(k\sqrt{nk}\log^2 n)$ for the queries plus $O(n\log^3 n)$ for (re)building the dense distance graph. By choosing k to be $n^{1/3}\log^{2/3} n$ we get an amortized complexity of $O(n^{2/3}\log^{7/3} n)$ per operation.

3.6.3 Dynamic algorithms for graphs with negative edge weights

We follow the same strategy in this case, as well. That is, we simply maintain the notion of the activated graph during a sequence of k updates. To answer a query for a pair (u, v), we compute a distance labelling in the extended activated graph for (u, v).

Unfortunately, there may be negative edges in the extended activated graph so we cannot just do a Dijkstra computation as above.

We note that if we have a feasible price function over the node set of the extended activated graph, if only one edge $e = (u, v)$ is updated with a negative weight w, we can use one computation of Dijkstra's algorithm to update the price function as follows. We compute the shortest distance labels $d(\cdot)$ of all the nodes starting from v. If $d(u)$ is greater than $-w$, changing the weight of e does not introduce any edge with a negative reduced cost on the graph with $d(\cdot)$ as a price function; hence, we can update e and update the price function to be $d(\cdot)$.

Therefore, if we already have k updates, we can compute a feasible price function in the extended activated graph by performing k Dijkstra computations by starting with the original price function on the extended activated graph, and for each update, we update the price function as described above.

After we have the feasible price function for the extended activated graph which includes all the updates, we can proceed as in the previous section.

After k queries and updates, we rebuild the dense distance graph. Thus, the total time for a sequence of k queries and updates is $O(k^2\sqrt{nk}\log^2 n)$ for recomputing the price function and $O(n\log^3 n)$ for (re)building the dense distance graph. By choosing k to be $n^{1/5}\log^{2/5} n$, we get an amortized complexity of $O(n^{4/5}\log^{13/5} n)$ per operation.

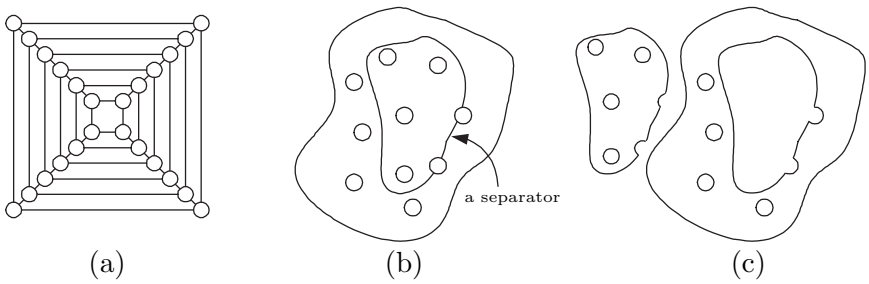


Figure 3.4: (a) An example of a graph which cannot be decomposed without introducing a hole. (b) a simple cycle separator dividing the holes between two subpieces. (c) the two subpieces of the decomposition.

3.7 Dealing with holes

It is not difficult to work with a piece with holes if only a constant number of holes are present. We will describe the algorithm for finding such a decomposition in subsection 3.7.1. Here, we continue to discuss how one can modify the algorithm given the required decomposition.

We assume that there are at most h holes on each piece. The on-line bipartite Monge data structure in section 4.3 can be modified to work with edges going between a pair of holes. In this case, we need to consider the ordering of nodes circularly, which can be done by allowing the intervals of left-side nodes to be able to wrap around. Therefore, we need at most h^2 data structures for edges in this category.

For edges between pairs of border nodes on the same holes, we use another h on-line Monge data structures.

The running time for data structure operations increases by at most a factor of h^2, which is a constant. Therefore, if every piece has $O(1)$ holes, we can implement the algorithm to run in the same time bound.

3.7.1 Graph decomposition with a constant number of holes

As mentioned in section 3.3, we cannot always have a decomposition that introduces no holes in all pieces (see figure 3.4 (a), for example), however it is possible to keep the number of holes in each piece below some constant h.

In this section, we give a procedure that decomposes the graph such that for every

constant number of levels of the decomposition, the numbers of nodes and border nodes on each piece go down geometrically while the number of holes remains at most a constant.

We use the simple cycle separator algorithm of Miller [Mil86] to divide each piece into smaller ones. The algorithm takes as an input a planar graph G with constant face size containing n nodes with node weights, and finds a simple cycle separator of size $O(\sqrt{n})$ which separates the graph into two subgraphs, each of which contains at most 2/3 of the total weight.

We show how to use the simple cycle separator algorithm to divide the graph so that we introduce at most one hole each time. We assume first that the graph is connected. To get the face size to be a constant, we triangulate the graph. The edges added during the triangulation process allow the cycle to jump over some faces, and because of these edges, the resulting subgraphs might contain many connected components. However, at most one hole gets introduced to at most one of the subgraphs.

For the piece with many connected components, we will apply Miller's algorithm on at most one of the components; hence, at most one hole is created. We proceed as follows. We note that if no components have more than 2/3 of the weights, we can divide the piece, without introducing any separator nodes, into two subpieces each with at most 2/3 fraction of the total weight. Now assume that there exists a unique component C that has more than 2/3 of the weights. We run the Miller algorithm only on C, introducing at most one hole. Now, we are left with components each having at most 2/3 fraction of the weights, so we can divide them without introducing new separator nodes.

With different weight assignments, we can divide the piece so that the subpieces satisfy various constraints. It is a standard application of the algorithm to divide the graph so that the number of nodes or border nodes in the subpieces drops by a factor of 2/3.

To divide the piece so that the number of holes decreases, we contract the border nodes on each hole into a super node and place uniform weights on the super nodes. After applying the Miller algorithm, each subgraph contains at most a 2/3 fraction of all super nodes. In other words, it contains at most 2/3 of the original holes. For any super node on the separator, the hole corresponding to that super node is broken into two parts and the border nodes of that hole become parts of the newly introduced hole. However, we introduces at most one hole during the process. Figures 3.4 (b) and (c) illustrate the way we divide the holes.

We now describe how to obtain a decomposition with the claimed property. We divide

a piece in the way that depends on which level of the decomposition it belongs to. More specifically, at level $3i$ we reduce the number of nodes, at level $3i+1$ we reduce the number of border nodes, and at level $3i+2$ we reduce the number of holes.

On every $3i$-th level, the numbers of nodes decrease by a factor of $2/3$. At each level we introduce at most new $O(\sqrt{n})$ border nodes. The number of nodes in each subpiece decreases geometrically, and the number border nodes in each subpiece with n nodes is at most $O(\sqrt{n})$.

On each level that we decompose a piece, we introduce at most one new hole. Since for every 3 levels, the number of holes drops by a factor of $2/3$, the number of holes remains at most a constant.

The Miller algorithm runs in linear time; thus, the time for decomposing the graph is $O(n \log n)$.

Chapter 4

Monge searching data structures

In this chapter, we describe the data structure that underlies the shortest path algorithms. We describe the general setting of the bipartite Monge searching problem in section 4.1. We develop an on-line version of the data structure in section 4.2. Finally, in section 4.3, we show how to use the on-line bipartite data structures to build the data structure for on-line non-bipartite Monge searching which is used in the implementation of Dijkstra's algorithm.

We note that the interface of the non-bipartite data structure is rather involved. Also, the technique for reducing the general case to the bipartite case is essentially the same as in the edge relaxation step in our implementation of Bellman-Ford.

4.1 Bipartite Monge searching

Given ordered sets A and B and a distance function $d : A \times 0B \to \mathrm{R}$ between pairs of element in A and B, we say that d has the *Monge property* if for all $u, v \in A$ and $x, y \in B$, $u \leq v$ and $x \leq y$ implies that

$$d(u, x) + d(v, y) \leq d(v, y) + d(u, x),$$

i.e., the sum of the distances when the pairs do not cross is at most the sum when the pairs cross (see Figure 4.1(a)). We can also view the triplet (A, B, d) as a metric on a complete "ordered" bipartite graph. Naturally, we call each element in A and B a node. Nodes in A are *left nodes*, and nodes in B are *right nodes*.

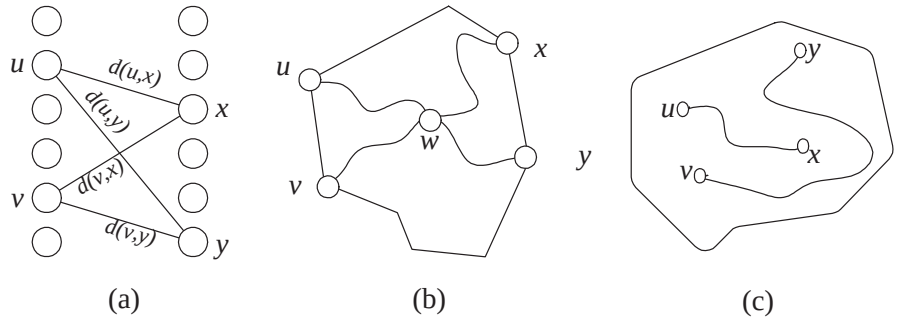


Figure 4.1: (a) The pairs of distances, (b) the Monge property for distances in planar graphs, (c) when two paths do not cross.

An example of distance functions with the Monge property is the shortest distances between border nodes in planar graphs. Figure 4.1(b) gives an example. Since any paths from u to x and from v to y always cross at some node w, we can break up the distance on the left hand side and derive the inequality.

The Monge property also implies another important property that we use. That is, for $y \in B$, if $u \in A$ is the node that minimizes $d(u, y)$, for any $v \geq u \in A$ and $x \leq y \in B$, $d(v, x) \geq d(u, x)$. Figure 4.1(b) also demonstrates this fact in the case of distances in planar graphs. To see this, note that the paths u-y and v-x must cross at some node w, whose distance label must be either from u or v. However, it must be the case that the distance from u to w is no worse than the current label of w because u minimizes $d(u, y)$. This implies that $d(u, x)$ is no worst than $d(v, x)$. The requirement that this two paths cross is very crucial, as this fact is not true in the example in Figure 4.1(c). Intuitively, this fact says that shortest paths need not cross. We call this property a *non-crossing* property. We prove the general case in the following lemma.

Lemma 4.1 *Suppose that the metric d on a complete bipartite graph (A, B) has the Monge property. If $u \in A$ minimizes $d(u, y)$ for some $y \in B$, then for all $v \geq u \in A$ and $x \leq y \in B$,*

$$d(v, x) \geq d(u, x).$$

Proof: From the Monge property, we have $d(u, y) + d(v, x) \geq d(u, x) + d(v, y)$. Since u minimizes $d(u, y)$, $d(u, y) \leq d(v, y)$, and it follows that $d(v, x) \geq d(u, x)$. ∎

Given (A, B, d) with the Monge property, the *Monge searching problem* is to find a parent $p(x) \in A$ for all $x \in B$ such that $d(p(x), x) \leq d(u, x)$ for any $u \in A$. The lemma

Algorithm B-MongeSearch($A = \{a_1, a_2, \ldots, a_n\}, B = \{b_1, b_2, \ldots, b_m\}, d$)

1. if $m = 0$, return
2. if $n = 1$

2.1 let $p(b_i) \leftarrow a_1$ for all $1 \leq i \leq m$

3. else

3.1 $mid \leftarrow \lceil m/2 \rceil$

3.2 let i be such that $d(a_i, b_{mid})$ is minimized.

3.3 $p(b_{mid}) \leftarrow a_i$.

3.4 B-MongeSearch($\{a_1, \ldots, a_i\}, \{b_1, \ldots, b_{mid-1}\}, d$)

3.5 B-MongeSearch($\{a_i, \ldots, a_n\}, \{b_{mid+1}, \ldots, b_m\}, d$)

Figure 4.2: Algorithm B-MongeSearch.

above states that it is enough to look at function $p(\cdot)$ that has no "crossing." Formally, it is sufficient to consider only function $p(\cdot)$ such that for $x \leq y$, $p(x) \leq p(y)$. Therefore, we can devise a simple divide-and-conquer algorithm, as shown in Figure 4.2, for finding p which runs in time $O(|A| \log |B|)$.

The idea is to find the correct parent for the middle right node first by checking all the left nodes, and then recurse on the top half and the bottom half of the right nodes. Each left node is to be considered at most twice for each recursive level.

4.2 On-line bipartite Monge searching

Suppose that a distance function d on a bipartite graph (A, B) has the Monge property. Given any function $D : A \rightarrow \mathrm{R}$, the distance function over edges $d'(u, x)$ defined to be $D(u) + d(u, x)$ also has the Monge property. Later on, we call D *distance labels* on the left nodes.

If the distance labels are completely specified, we are in the bipartite Monge searching problem. In the on-line version, the distance labels are not specified initially. During the life of the data structure, the user keeps "revealing" the distance label of some left node. In the *on-line bipartite Monge searching problem*, we want to maintain the best parents $p(x)$ for all $x \in B$ as the distance labels $D(\cdot)$ of nodes in A get specified.

Furthermore, we also want to be able to answer a query that asks for the right node

with the minimum distance after each update. This is for the implementation of Dijkstra's algorithm.

In section 4.2.1, we describe a data structure that maintains the best parent for each right node. We show how to extend the data structure to support the minimum distance queries in section 4.2.2.

4.2.1 Maintaining best parents

Formally, we are given a bipartite graph $G = (A, B, E)$ with a distance function d over edges satisfying the Monge property and distance labels $D(\cdot)$, not completely specified, for nodes in A. The cost for an edge (u, x) is now $D(u) + d(u, x)$. Without loss of generality, for any left node u with unspecified distance label, we let $D(u) = \infty$. Let $n = \max(|A|, |B|)$. We label all the left nodes as $a_1, a_2, \ldots, a_n$ and all right nodes as $b_1, b_2, \ldots, b_n$ according to their order.

We first describe the operations that we want to supports.

- B-Activate(u, d_u). For a left node $u \in A$, set $D(u)$ to be d_u.
- B-FindParent(x). For a right node x return $p(x)$.

From lemma 4.1, we can consider only p which forms a non-crossing structure (see Figure 4.3). We maintain a searchable ordered list $P = [a_{i_1}, a_{i_2}, \ldots, a_{i_k}]$ of indices of all left nodes which are the best parents for some right node. For each node $a_j \in P$, we maintain two pointers $s[a_j]$ and $t[a_j]$ to nodes in B such that a_j is the best parent for a set of right nodes $s[a_j] = b_p, b_{p+1}, \ldots, b_q = t[a_j]$. We denote this set of nodes by $B(a_j)$ and call it an *interval* for a_j. We note that the family of intervals $\{B(a_j) : a_j \in P\}$ partitions B.

With this data structure, one can implement B-FindParent in $O(\log n)$ time by searching into P. We need, however, to maintain the list and the pointers s and t for each node in the list when some left node u is activated. To do so, we find the interval for u and update the interval data structure accordingly. The idea is as follows. By comparing the distance to some right node x from u and its current parent $p(x)$, we can decide if the interval for u lies above or below x. Then the actual interval can be found using binary search. Note that basically, $O(\log n)$ comparison is needed; for each right node x in one comparison, finding $p(x)$ takes $O(\log n)$ time, resulting in a running time of $O(\log^2 n)$.

The running time for finding the interval for u can be further improved. We show how to find the beginning of the interval $s[u]$ for u; $t[u]$ can be found using the same idea. First

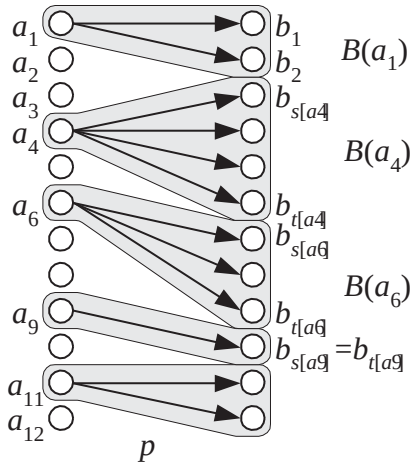


Figure 4.3: Non-crossing structure of $p(\cdot)$.

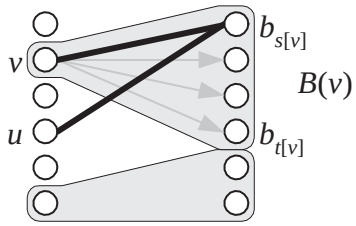


Figure 4.4: If $D(u) + d(u, s[v]) < D(v) + d(v, s[v])$, u takes v's entire interval.

note that if for some parent $v \leq u$, $s[v]$ is in u's interval, every right node in v's interval must now be in u's interval (see Figure 4.4), and the interval for v is removed forever. Thus, the time spent with v can be charged to the time v was created. Now, we can sequentially search for the parent w such that $s[u]$ lies strictly in its interval. The time for the sequential search is charged to the time the intermediate intervals were created. To find the exact node $s[u]$ in $B(w)$, we can now use binary search in $O(\log n)$ time. Therefore, we can find the intervals for l activated nodes in $O(l \log n)$ time.

After finding the interval for u, we need to update the data structure. For all the intervals which are removed, we charge the running time to the time it was created. At most two intervals need to be modified. Another interval is created for u and inserted into the list P. Figure 4.5 shows the interval structure after change. All these operations can be done in $O(\log n)$ time. Thus, we have that the time for updating after l calls to B-Activate

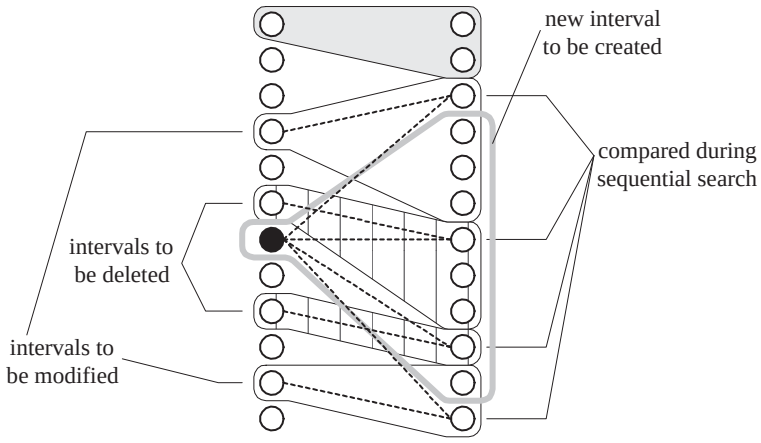


Figure 4.5: A new interval is created, at most 2 are modified, while a few might be deleted.

is $O(l \log n)$ time.

We can implement the operations as shown in Figure 4.6. We begin by using P to find two closest parents a_{i_j} and $a_{i_{j+1}}$ in $O(\log n)$ time. Then, SearchUpward finds $s[u]$ of the interval for u by sequential searching for parent v where $s[u]$ lies and binary searching in v's interval to find $s[u]$ (in procedure FindUpperRange, which is not shown here). We also call SearchDownward to find $t[u]$. The time for the sequential searches can be accounted for, and the time for two binary searches is $O(\log n)$. Then, we update P and all the pointers in $O(\log n)$ time. Therefore, we have to following lemma.

Lemma 4.2 *We can implement the data structure so that* B-FindParent *runs in* $O(\log n)$ *time and* l *calls to* B-Activate *run in* $O(l \log n)$ *time.*

We note that this data structure also supports many calls to B-Activate for a single left node, provided that the distance labels specified are non-increasing over time.

4.2.2 Answering queries

We want to use this data structure to implement Dijkstra's algorithm as described in section 3.4.3. We start by explaining briefly how the data structure is used in the algorithm. We partition the set of border nodes in the dense distance graph into 2 sets, the set of left nodes A and the set of right nodes B, and consider all edges going from A to B. As in Dijkstra's algorithm, we maintain a set $S \subseteq (A \cup B)$ of scanned nodes. We want to relax all

Algorithm B-FindParent($x = b_l$)

1. find $a_i \in P$ such that $s(a_i) \leq x \leq t(a_i)$
2. return a_i

Algorithm B-Activate($u = a_l, d_u$)

1. $D(u) \leftarrow d_u$
2. find a_i and a_j which lie consecutively in P such that $a_i < u < a_j$
3. $s[u] \leftarrow$ SearchUpward(u, d_u, a_i)
4. $t[u] \leftarrow$ SearchDownward(u, d_u, a_j)
5. if $s[u] \leq t[u]$ then /* u is a parent for some right nodes */

5.1 $a_q \leftarrow p(s[u])$ /* these two intervals ($B(a_q)$ and $B(a_r)$) need to be updated */

5.2 $a_r \leftarrow p(s[t])$

5.3 remove all intervals in P after a_q and before a_r

5.4 insert u into P at the correct position, i.e., after a_q

5.5 $t[a_q] \leftarrow$ Prev$(B, s[u])$

5.5 $s[a_r] \leftarrow$ Next$(B, t[u])$

Algorithm SearchUpward(u, d_u, a_j)

1. $v \leftarrow a_j$
2. while $v \neq nil$ and $d(u, s[v]) < d(v, s[v])$ do

2.1 $v \leftarrow$ Prev(P, v)

3. if $v \neq nil$ then

3.1 return FindUpperRange(u, v)

4. else

4.1 return b_1

Figure 4.6: Pseudocodes for algorithms B-FindParent and B-Activate.

these edges quickly when a left node is scanned. We also need to find the unscanned right node with the minimum distance label.

Let denote by S_B the set $S \cap B$ of scanned right nodes. To motivate the idea, we first consider the simpler case where S_B remains empty, i.e., none of the right nodes has been scanned. Therefore, we only have to keep track one minimum node over all the right nodes. This can be accomplished by using the range search tree data structures. For each left node a_i, we build a range search tree $T_I[a_i]$ for the distances for edges (a_i, b_j) for all $b_j \in B$. Now, when a node is activated, we use T_I to find the minimum nodes within the modified intervals and update the minimum node[1].

Now, when B_S is not empty, for each modified intervals, it is not clear that finding the minimum node in $B \setminus B_S$ can be done by one search in T_I, because, in general, the interval of a left parent u might be fragmented, i.e., $B(u) \setminus B_S$ contains many sub-intervals (see Figure 4.7). However, in our application, we can exploit the way we put any right node into B_S. Note that when any node is scanned, its distance label is the minimum among the unscanned nodes, and it remains the minimum among these nodes through out the execution of the Dijkstra's algorithm. This implies that, in our terminology, when a right node x is scanned, no intervals will ever cross the edge $(p(x), x)$ again. This ensures that the interval structure, even with the set B_S, remains non-crossing.

From the above discussion, we want extend the Monge searching data structure from the previous section to be able to maintain the set $S_B \subseteq B$, initially empty, and support these operations:

- B-Activate(u, d_u). For a left node $u \in A$, set $D(u)$ to be d_u.
- B-FindParent(x). For a right node x return $p(x)$.
- B-FindMin(). Return the right node in $B \setminus S_B$ with the minimum distance label, together with its parent.
- B-ExtractMin(). Return $v =$ B-FindMin() with its parent and set $S_B \leftarrow S_B \cup \{v\}$. Upon calling this operation, the user guarantees that (1) distance label of v, which is the current minimum node, will not change later on, and (2) among all the unscanned

[1]In fact, if there is no multiple calls to B-Activate for a single node, we only need to find the minimum right node only for the recently activated node, because the distance labels of minimum right nodes in other intervals are non-decreasing.

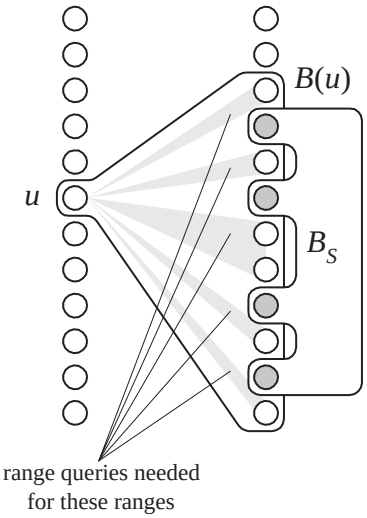


Figure 4.7: In general, $B[u] \setminus B_S$ can have many intervals.

right node at the time of call the distance label of v remains the minimum through the entire execution of the data structure.

The data structure keeps the following.

- For each left node u, a range search tree $T_I[u]$ that keeps all edges (u, x), for all $x \in B$, and supports queries for the minimum edge length over ranges. These trees are constructed *a priori* in time linear in the number of edges in the bipartite graph.
- For each interval $B(u)$, the node $c[B(u)] \in (B \setminus B_S)$, a node with the minimum label in the interval, which can be found using the range search tree $T_I[u]$.
- A priority queue H for these minimum node for all the intervals in P.

It is simple to modify B-Activate to incorporate these changes. Also, B-FindMin can be implemented by returning the first element in queue H. Now, We only need to show how to implement B-ExtractMin.

From the previous discussion, the constraint we impose upon calling B-ExtractMin ensures that if B-FindMin returns x, no intervals will cross $(p(x), x)$ again. To take care of this B_S matter, we want x to be out of any query computations later on. Therefore, we do as follows. We break the interval $B(p(x))$ into 3 intervals (see Figure 4.8). The middle

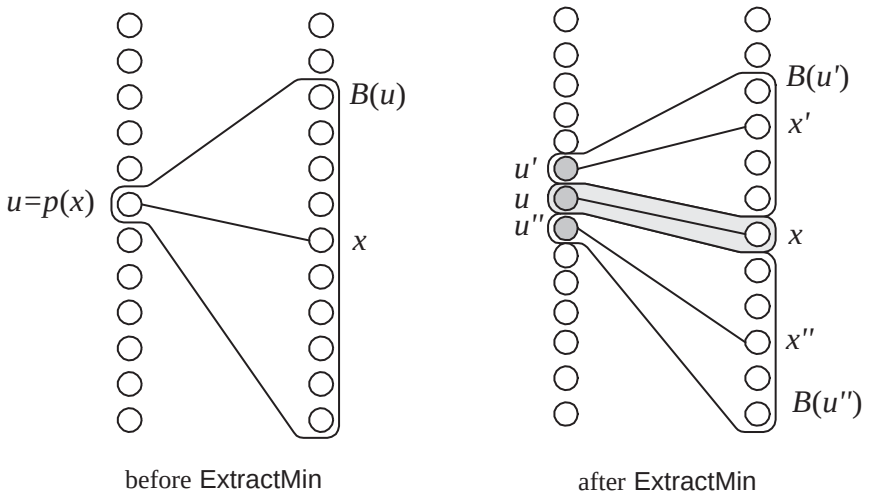


Figure 4.8: $B[u]$ gets split after B-ExtractMin.

interval is the one containing only $(p(x), x)$. The first interval contains nodes before x, and the second interval contains nodes after x in B. From the choice of x, the middle interval will never change. For each of the newly created intervals, we make a copy of the parent $p(x)$ and insert it to P accordingly. This might create new left nodes, but we will never create more than $2n$ left nodes.

The pseudo codes for B-FindMin and B-ExtractMin are shown in Figure 4.9.

Since in steps 6 and 7 of B-ExtractMin, we use the interval trees $T_I[u']$ and $T_I[u'']$[2], the running time for B-ExtractMin is $O(\log n)$. Note that B-FindMin runs in time $O(1)$, even though B-FindParent runs in $O(\log n)$ time, because we keep the minimum right nodes along with their parents in the priority queue H. parent. Also the modified B-Activate has the same time complexity. We summarize the running time for this bipartite Monge search data structure in the following lemma.

Lemma 4.3 *The data structure for the bipartite Monge searching problem can be implemented so that*

- *l calls to* B-Activate *run in $O(l \log n)$ time,*
- B-FindParent *and* B-ExtractMin *run in $O(\log n)$ time, and*

[2]They point, however, to the same interval tree.

Algorithm B-FindMin()

1. return the minimum element in H.

Algorithm B-ExtractMin()

1. $(u, b_j) \leftarrow deleteMin(H)$
2. create left nodes u' and u''
3. let $T_I[u'] = T_I[u''] = T_I[u]$
4. $s[u'] = s[u]; t[u'] = b_{j-1}$
5. $s[u''] = b_{j+1}; t[u''] = t[u]$
6. $c[B(u')] \leftarrow$ minimum node in the range $s[u'] \ldots b_{j-1}$
7. $c[B(u'')] \leftarrow$ minimum node in the range $b_{j+1} \ldots t[u'']$
8. insert $(u', c[B(u')])$ and $(u'', c[B(u'')])$ to H
9. insert u' and u'' into P

Figure 4.9: Pseudocodes for algorithms B-FindMin and B-ExtractMin.

- B-FindMin *runs in* $O(1)$ *time.*

4.3 Non-bipartite on-line Monge searching

Given a graph $G = (V, E)$ whose nodes are in a circular order, with the distance function d on the edges, we say that d satisfies the Monge property if

$$d(u, w) + d(v, x) \geq d(u, x) + d(v, w),$$

for every $u, v, w, x \in V$ such that $u \leq v \leq w \leq x$ in V. Note that the sign of the inequality is different from the bipartite case because now (u, w) crosses (v, x).

In this section, we consider the *non-bipartite on-line Monge searching problem.* We are given a graph $G = (V, E)$ with the distance function d satisfying the Monge property. We also have distance labels D over the nodes, which are to be specified by the user during the running of the data structure. The distance labels D and the distance function d induce the *internal* distance label on a node v, which is defined to be $D'(v) = \min_{u \in V} D(u) + d(u, v)$. To avoid confusion, we will call the distance labels D *starting* distance labels. We also maintain a set S of nodes. In this problem, we want to

- maintain the parent structure of nodes p, i.e., for any nodes $v \in V \setminus S$, $p(v)$ is the node u that minimize $D(u) + d(u, v)$, and
- report the node $v \in V \setminus S$ with the minimum internal distance label $D'(v)$.

We note the difficulties in this case. In the bipartite case, the set of nodes which has a particular node as their parent is consecutive, i.e., that particular node owns a single interval. This is not true in the non-bipartite case. However, we show in this section how to reduce this problem to $O(\log n)$ bipartite cases. The idea is to partition the edges as in section 3.4.2.

Let $k = 2\lceil \log n \rceil$. From G, we create k bipartite graphs, because for each left-right partition, edges between them can go in two directions. We denote these bipartite graphs as $G_0, G_1, \ldots, G_k$. Under this reduction, each edge belongs to one and only one bipartite graph. We refer to each bipartite graph G_j as the level j bipartite graph of G. Each node v now belongs to the sets of right nodes in $O(\log n)$ levels. We call the distance label of v on each of these levels the internal distance label *on* that level. The internal distance label of v is the minimum internal distance label over all the levels.

We formalize the interface of this data structure. We maintain the starting distance label D over the nodes and a set S of nodes. Ideally, we want the data structure to provide the following operations.

- Activate(v, d_v). For a node $v \in V$, set $D(u)$ to be d_v.
- FindParent(v). Return $p(x)$.
- FindMin*(). Return the node in $V \setminus S$ with the minimum internal distance label, together with its parent.
- ExtractMin*(). Return $v =$ FindMin() with its parent and set $S \leftarrow S \cup \{v\}$.

However, there is some technical detail. Recall the requirement that, for B-ExtractMin to work correctly, the distance label on the minimum right nodes, which is to be returned by the procedure, must remain the same and remain the minimum over all the unscanned nodes at that time of that call. Since we use the on-line bipartite data structures here, we also need the same condition. Assuming that v is the node which is returned by FindMin, i.e., v's internal distance label on some level i is the minimum, we want the internal distance label of v on level i to remain the same and remain the minimum over all internal distance

labels of every right node on level i. Note that the way that we scan nodes implies that the condition is true. We therefore change FindMin* and ExtractMin* to the following.

- FindMin(). Return a node v, together with its parent, such that the internal distance label $D'(v)$ of v is smaller than any nodes in $V \setminus S$. The node v returned by this operation might already be in S.

- ExtractMin(). Return $v =$ FindMin() with its parent and set $S \leftarrow S \cup \{v\}$. Note that v can be returned by ExtractMin again even when it is in S. However, we guarantee that v can be returned by this operation for at most $k = O(\log n)$ times, i.e., after $O(\log n)$ calls to ExtractMin for some node, FindMin will never return that node again.

  We further require the same condition for the distance label of the current minimum node v as in B-ExtractMin; i.e., (1) the internal distance label of v on that level will not change later on, and (2) among all the unscanned right nodes on that level at the time of the call, the internal distance label of v on that level remains the minimum through the entire execution of the data structure.

The data structure can be implement directly from k bipartite data structures. When Activate(v) is called, we call the bipartite B-Activate(v) for all the $k/2$ levels that v belongs to the left nodes. For FindParent(v), we examine all the best parents for v in all the levels and return the best one.

We maintain a priority queue H' of the minimum nodes over all the levels, and use this to return the minimum node in FindMin. To implement ExtractMin, when a node v from the level j bipartite graph is the node with the minimum internal distance label, we only call the bipartite B-ExtractMin only in the level j bipartite graph.

The next theorem, the main theorem for this chapter, shows the running times.

Theorem 4.4 *In the non-bipartite Monge searching problem, we have a data structure that supports*

- l *calls to* Activate *in* $O(l \log^2 n)$ *time,*
- FindParent *in* $O(\log^2 n)$ *time,*
- FindMin *in* $O(1)$ *time, and*
- ExtractMin *in* $O(\log n)$ *time.*

Furthermore, any node v will never be returned by ExtractMin *more than* $O(\log n)$ *times.*

Proof: Activate can be done by calling the bipartite B-Activate on all the levels to update the parent structure. On these levels, we call the bipartite B-FindMin and update their entries in the priority queue H'. Since l calls to the bipartite B-Activate on each level take $O(l \log n)$ time and we have $k = O(\log n)$ levels, l calls to Activate take $O(l \log^2 n)$ time. The time to update the priority queue is $O(\log n \log \log n)$ because H' has at most $O(\log n)$ elements.

Clearly FindMin can be done in $O(1)$ time by returning the minimum entry in the heap H'.

For ExtractMin, we call FindMin which return a node from some level i bipartite graph, then we call the bipartite B-ExtractMin on that bipartite graph. We then call the bipartite B-FindMin in the same bipartite graph and update H'. The running time for this operation is, thus, $O(\log n)$.

Note that each node belongs to at most $k = O(\log n)$ levels. For each level, after a node is returned by B-ExtractMin, it can never be returned again. Therefore, any node can not be returned, over all the levels, for more than k times. ∎

Chapter 5

Edge-disjoint paths in planar graphs

Various problems in computer networks, e.g., path selection, routing, network design, can be modeled using a capacitated graph with a set of k source-sink pairs, or commodities. The basic problem is to find a set of paths which connects all the source-sink pairs such that the paths mutually share no edges. It is not clear whether it is possible to do so. In fact, the problem of determining if the set of paths exists is one of NP-complete problems in Karp's seminal work [Kar72]. It remains NP-complete when the graph is planar [Lyn75] and even a 2-dimensional mesh [KvL84]. For directed graph the problem is NP-hard when $k = 2$ [FHW80]. In undirected graphs, the situation is much better. Robertson and Seymour [RS95] gave an $O(|V|^2 \cdot |E|)$ time algorithm for the problem when the number of commodities is fixed. The constant in the running time, however, depends severely on k. The quest for sufficient and necessary conditions for the existence of the disjoint paths motivated, and continues to motivate, the research in combinatorics and computer science; for the history and basic results, we direct the reader to a survey by Frank [Fra90].

One of the natural necessary conditions that we investigate in this chapter is the *cut criterion*, which states that the capacity, i.e., the number of edges, of every cut must be at least the number of commodities whose sources and sinks are separated by the cut. This condition is not sufficient in general. However, in many cases, it is. Okamura and Seymour [OS81] showed that in a planar graph where every source and sink lies on the boundary face, the cut criterion is sufficient provided that some degree constraint is satisfied.

(See, also, Wagner and Weihe [WW95] for a linear-time algorithm and a different proof of the theorem.)

The result we present is a generalization of the Okamura-Seymour theorem. We consider the case for a planar graph where every sink lies on the boundary face. In this case, the cut criteria is not sufficient. However, if each edge is allowed to be used for a constant number of times, we show that the cut criteria is a sufficient condition and give an algorithm for finding the set of paths. We also give a polynomial time algorithm that find the sparsest cut, defined to be the worst cut in terms of the ratio between the capacity and the number of commodities crossing the cut.

5.1 The problem

Given a connected graph $G = (V, E)$ and a set of k pairs of nodes $(s_1, t_1), \ldots, (s_k, t_k)$, the *edge-disjoint path problem* is to find a set of pairwise edge-disjoint paths $p_1, \ldots, p_k$ that connects all the demands, i.e., for all i, $p_i \ni s_i$ and $p_i \ni t_i$. We can form a graph H from the set of pairs (s_i, t_i). An instance of the edge-disjoint path problem can then be represented by a pair of graphs (G, H). We call G the *supply graph* and H the *demand graph*.

One of necessary conditions for the existence of edge-disjoint paths is called the *cut criterion*. For any subsets of nodes X and Y in G, we denote by $d_G(X, Y)$ the number of edges with one end in $X - Y$ and one end in $Y - X$. We also write $d_G(X)$ for $d_G(X, V - X)$. If an edge e has one end in X and another end in $V - X$, we say that e *crosses* X; hence, $d_G(X)$ is the number of edges crossing X. The cut criterion is defined as follows:

$$d_G(X) \geq d_H(X),$$

for all $X \subset V$. This condition states that the number of demands crossing the cut is at most the capacity of the cut.

The cut criterion is known to be insufficient (for examples, see Figure 5.1). However, in planar graphs, Okamura and Seymour [OS81] showed that if all sources and sinks are on the same face and all nodes in $G + H$ have even degree, the cut criterion is sufficient.

For a cut X, the ratio $\frac{d_G(X)}{d_H(X)}$ is call the *sparsity* of X. Another problem which is related to the edge-disjoint path problem is the *sparsest cut problem* where we want to find the cut X with the minimum sparsity, called the *sparsest cut*. By using appropriate demand graphs,

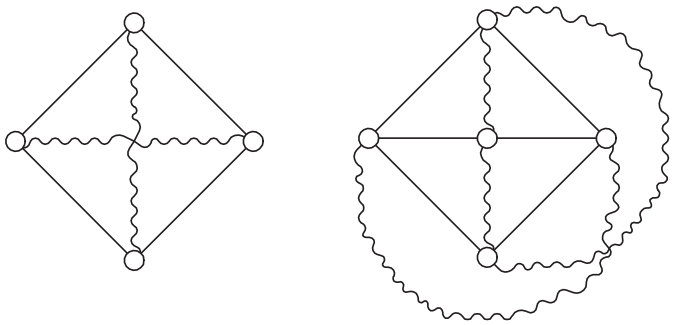


Figure 5.1: Two examples for the cases where cut criteria hold, but no edge-disjoint paths exist. The demands are shown in curly lines. In (a), half-disjoint paths exist, but for (b), even fractional paths do not exist.

the algorithm for this problem can be used as a subroutine to find balanced cuts [LR88], which is a crucial step in many divide-and-conquer-type algorithms.

5.1.1 Our result

In this chapter we study the approximate relation between the cut criterion and the existence of edge-disjoint paths in planar graphs. We relax the requirement so that the paths need not be mutually edge-disjoint by allowing them to use some edge many times. We then measure how good any solution is by the maximum number of times any edge is used, defined to be the *congestion* of this set of paths.

In section 5.2, we give a generalization of the Okamura-Seymour theorem to the cases where all the sinks lie on the same face. More specifically, we show that if the cut criterion holds, a set of paths with congestion at most 5 can be found in polynomial time. Furthermore, if half-integral paths are allowed, we can find a set of paths with congestion 3.

We show that in this case we can also find the sparsest cut in polynomial time in section 5.3.

5.1.2 Related results

For undirected planar graphs, the long line of research on the disjoint paths problem began with many restricted classes of planar graphs, e.g., trees [RU94], trees with

parallel edges [GVY97], two-dimensional meshes [AGLR94, AR95], and was extended to include larger classes of planar graphs called densely embedded graphs by Kleinberg and Tardos [KT95], where various constant factor approximation algorithms are shown.

In general directed graphs, although this problem is NP-hard for $k = 2$, Fortune, Hopcroft, and Wylie [FHW80] showed that for a fixed value of k, the problem is solvable in polynomial time for acyclic graphs. Schrijver [Sch93, Sch94] showed the same fact for planar directed graphs.

We point the reader to Jon Kleinberg's thesis [Kle96] for the history, references, and recent results on the disjoint path problem.

The strict requirements for the edge-disjoint path problem can be relaxed, basically, in two ways. Firstly, we can allow fractional paths (in other words, fractional flows). Secondly, we can allow multiple, but preferably a small number, uses of each edge, i.e., we want to minimize the congestion. This relaxed version of the problem, called *the concurrent flow problem* first defined by Shahrokhi and Matula [SM90], can be solved or approximated in polynomial time (using various algorithms, e.g., [SM90, PST91, GK98, KP95]). However, the crux is to find a way to use the fractional solutions. Along this line, many important results and techniques were discovered including the randomized rounding technique of Raghavan and Thompson [RT87] and the approximate max-flow min-cut theorem of Leighton and Rao [LR88, LR99].

The sparsest cut problem is closely related to the problem of finding a balanced cut, which is an important step in a number of problems. Rao [Rao87] was the first to focus on this problem, however, in the context planar graphs. The problem in general graphs was taken by Leighton and Rao [LR88, LR99]. They showed that the approximate max-flow min-cut ratio between the sparsest cut and the value of fractional flows is $\Theta(\log n)$ in the case of uniform demands on an n-node graph. This result provides a starting point for numerous divide-and-conquer approximation algorithms on network problems. For general demands, the ratio has been proved and improved by various researchers [KRAR95, Tra91, Kah93, PT93]. The final ratio, based on embedding techniques, of $\Theta(\log k)$ is shown by Aumann and Rabani [AR98] and Linial, London, and Rabinovich [LLR95]. A survey by David Shmoys [Shm97] provides a useful starting point for the research on this problem.

For planar graphs, the conjecture is that this ratio is constant. The best known upper-bound of $O(\sqrt{\log n})$ for planar graphs was shown by Rao [Rao99] using the decomposition technique developed by Klein, Plotkin, and Rao [KPR93]. For many special cases, the ra-

tio is shown to be constant, e.g., for outerplanar graphs [GNRS99] and for k-outerplanar graphs [CGN+03].

5.2 Paths with constant congestion

Our result is a way to reduce the problem to the Okamura-Seymour case. Intuitively, we show that by "rerouting," one does not increase the number of commodities across any cuts by more than a factor of 2.

Consider a set of k mutually edge-disjoint paths $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ where each path P_i starts at s_i. Note that some P_i might be an empty path. Every path P_i ends at some node; denote it by w_i. From this set of paths, we know how to route every source s_i to w_i. Therefore, we can think of the new problem as to find a set of paths from intermediate nodes w_i to t_i. We modify the instance of the problem accordingly, i.e., for each demand (s_i, t_i), we replace it with (w_i, t_i). We then form the graph H' from the modified demands. If we can solve the new problem, the solution to the original problem can be recovered by concatenating the solution of the modified one with the set of paths $\mathcal{P}$. Since paths in $\mathcal{P}$ are mutually edge-disjoint, this increases the congestion by at most 1.

The main ingredient of our result is the following lemma, which states that after the rerouting, the cut criterion holds within a factor of 2.

Lemma 5.1 *If the cut criterion in the original problem (G, H) holds, the cut criterion in the problem $(2G, H')$ holds.*

Proof: Consider any cut S in G. Take any demand $d = (w_i, t_i)$ in H' that crosses S. Our charging scheme is as follows. If the original demand $d' = (s_i, t_i) \in H$ also goes across S, we charge d to d'. Otherwise, it must be the case that path p_i crosses S at some edge e; we charge d to e. The lemma follows by noting that the total demands not being charged to edges of the cut is at most the original demands in H which crosses the cut; hence, it is at most the capacity of the cut because the cut criterion holds in (G, H). ∎

We use the Okamura-Seymour theorem [OS81], which we state here.

Theorem 5.2 (Okamura-Seymour) *If the graph $G + H$ is Eulerian and every terminal lies on the same face, there exists a set of mutually edge-disjoint paths connecting the source-sink pairs if and only if the cut criterion holds.*

Wagner and Weihe [WW95] gave a linear time algorithm with an alternative proof of this theorem based on right-first search, a graph searching technique which exploits topological structure of the graph. We will use their algorithm for the implementation.

We are ready to prove the main result of this chapter.

Theorem 5.3 *If the cut criterion holds and every sink lies on the same face F, a set of paths with congestion at most 5 exists. Moreover, if paths are allowed to be half-integral, the congestion is at most 3.*

Proof: Let D' denote a set of all demands whose sources do not lie on the special face F. Since the cut criterion holds, if we treat these demands as the same commodity, the cut criterion holds still in this case. Therefore, from the max-flow min-cut theorem, we can perfectly match each source in D' with some sink in D' which lies on the face using a set of mutually edge-disjoint paths. We use this set of paths to route demands in D' to the face; we use each edge at most once here.

From lemma 5.1, the cut condition holds if we double all edge capacities. The problem is now applicable to the Okamura-Seymour theorem. If we are satisfied with half-integral paths, applying the theorem directly gives a set of paths which after combining with the paths from the first phases yields the solution with congestion at most 3. However, if integrality is required, we need that the graph $2G + H'$ is Eulerian. We can do so by double all edges and demands again, which leads to the congestion of 5 in the final solution. ∎

5.2.1 Implementation

The algorithm comprises of two phases. In the first phase, a set of paths for demands in D' is found in $O(n \log^3 n)$ time using an algorithm of Miller-Naor [MN95] with the shortest path algorithm we present in Chapter 3. Then, we use the algorithm by Wagner and Weihe [WW95] which runs in linear time in the second phase. Thus, the paths can be found in $O(n \log^3 n)$ time.

5.3 The sparsest cut

In this section, we give an algorithm that finds the sparsest cut in this special case. We start by stating a standard lemma on the submodular property of d. The following lemma, which can be proved by a simple counting argument, is very useful.

Lemma 5.4 *For any $X, Y \subset G$,*

- $d(X) + d(Y) \geq d(X \cap Y) + d(X \cup Y)$, *and*
- $d(X) + d(Y) \geq d(X - Y) + d(Y - X)$.

A cut *crosses* some face f when there is some edge adjacent to f whose end nodes are separated by the cut. The *number of times* the cut crosses f is the number of such edges. We are ready to prove a lemma that shows that the structure of the sparsest cut is simple.

Lemma 5.5 *If the sparsest cut crosses the face F, there exists a cut with the smallest sparsity which crosses F twice.*

Proof: Consider any cut C crossing F more than twice. First note that if the sparsest cut consists of many connected components, from the submodular property of the demand function d_H, there exists one with has only one connected component. Now the only case left is when the cut C is of the form $A_0 - A_1 - A_2 - \cdots - A_k$ where each A_i is a cut that crosses F only twice and for all $i > 0$, $A_i \subset A_0$ (see Figure 5.2). Consider the demands crossing C, we have that $d_H(C) \leq \sum_i d_H(A_i)$ from the submodular property of d_H. Since A_i's are mutually disjoint, we have $d_G(C) = \sum_i d_G(A_i)$. Thus,

$$\frac{\sum_{i=0}^{k} d_G(A_i)}{\sum_{i=0}^{k} d_H(A_i)} \leq \frac{d_G(C)}{d_H(C)}.$$

The lemma is proved by observing that there exists some j such that

$$\frac{d_G(A_j)}{d_H(A_j)} \leq \frac{\sum_{i=0}^{k} d_G(A_i)}{\sum_{i=0}^{k} d_H(A_i)} \leq \frac{d_G(C)}{d_H(C)}.$$

∎

Note that the sparsest cut need not cross the face F. We first deal with the case when it crosses the face. With the above lemma, we can enumerate all possible crossing points on the face. Let $e_1, e_2, \ldots, e_l$ denote a sequence of edges on the face F. We denote each crossing possibility by a pair (e_i, e_j), so there are $O(l^2) = O(m^2)$ possible crossing places. Note that we can constraint the cut to cross only at e_i and e_j by assigning infinite capacities to all other boundary edges (see Figure 5.3(a)).

Now, for each crossing place (e_i, e_j), the set of sources and sinks lying on F inside the cut are completely determined. Given that, we can set the weight $w(v)$ for each node v to

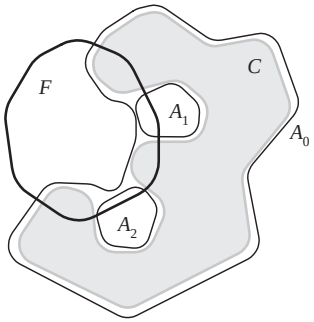


Figure 5.2: A cut with one connected component which crosses the face more than twice.

be the contribution to the number of demands crossing the cut if v is included. Specifically, for node v lying on the boundary of F inside the cut, i.e., those which are on the path from e_i and e_j, we let $w(v)$ be the number of commodities outside the cut having v as their terminals. For all other nodes v, we let $w(v)$ be the number of commodities outside the cut minus the the number of commodities inside the cut which have v as their sources. From this weight assignment, we note that the weight of some node can be negative. The problem becomes the problem of finding a cut C which crosses F at (e_i, e_j) that minimize the ratio

$$\frac{|\delta(C)|}{\sum_{v \in C} w(v)},$$

which is *the minimum-ratio cut problem*. This problem can be solved in polynomial time using the techniques developed by Rao [Rao92] and Park and Phillips [PP93].

We look at the problem in the dual; now the weights are on the faces. Denote by e_k^* the dual edge corresponding to e_k. We first define $G^*_{(e_i,e_j)}$ be the dual graph of G with the exception that for all edges e_k^*, for $k \notin \{i, j\}$, we set $c(e_k^*) = \infty$ to prevent the cut from crossing the face at these edges (see Figure 5.3(b), for example).

Take any spanning tree T in $G^*_{(e_i,e_j)}$ that contains all edges $e_1^*, e_2^*, \ldots, e_l^*$. Define, for each dual non-tree edge e^*, the enclosing weight $W(e^*)$ to be the sum of the weight inside the cycle induced by e^* and T.

To find the minimum-ratio cut, we start by guessing the ratio α, and build a directed graph H^α with length $l(\cdot)$ on the edges as follows.

The nodes of H^α are the nodes from G^*. For each dual non-tree edge e^* we create 2 arcs e_c^* and e_{cc}^*, going into opposite directions with respect to the root of the tree, with lengths $l(e_c^*) = c(e^*) + \alpha W(e^*)$ and $l(e_{cc}^*) = c(e^*) - \alpha W(e^*)$. (Figure 5.4 shows the example.) For

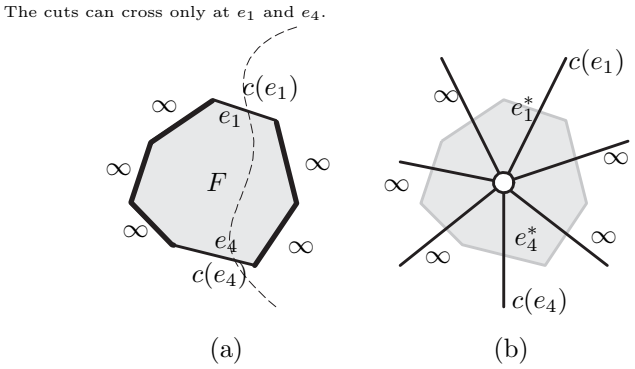


Figure 5.3: The modified graphs for when the crossing place is (e_1, e_4): (a) in the primal, (b) in the dual graph $G^*_{(e_1,e_4)}$.

all tree edges e^* in $G^*_{(e_i,e_j)}$, we create two arcs e_1^* and e_2^* in H^α with the same length $c(e^*)$. Under this reduction, for any cycle C' going counter-clockwise in H^α, the length of C' is $\sum_{e^* \in C'} l(e^*) = \sum_{e^* \in C'} c(C') - \alpha \sum_{v \in C'} w(v)$ (see Figure 5.5).

Using binary search, we can find α such that the length of the shortest cycles in H^α is zero. The following lemma states that if the sparsest cut crosses at (e_i, e_j), the procedure will find it.

Lemma 5.6 *Provided that all sparsest cuts cross the face, the above procedure finds the sparsest cut if there exists one crossing F only at (e_i, e_j).*

Proof: From the construction discussed above, given a parameter α, recall that the length of any counter-clockwise cycle crossing F at (e_i^*, e_j^*) is positive, zero, or negative, if the sparsity of the cut corresponding to it is less than, equal to, or greater than α, respectively.

Let C^* be the cycle of length zero found by the procedure. We assume first that C^* is counter-clockwise, and will consider the other case later.

Since C^* is counter-clockwise, the length of C^* equals $c(C^*) - \alpha(\sum_{v \in C^*} w(v)$. We know that it is zero, implying that we have $\alpha = c(C^*)/\left(\sum_{v \in C^*} w(v)\right)$. We let S be the cut in G corresponding to C^*. Since no other cycles have shorter length, we have that S is the cut that minimizes $d_G(S)/\left(\sum_{v \in S} w(v)\right)$. This ratio might not be the sparsity of the cut when it does not cross the face F, because the weight inside S need not be the number of demands crossing S. We consider it in two cases: (1) when C^* crosses F, and (2) when C^* does not.

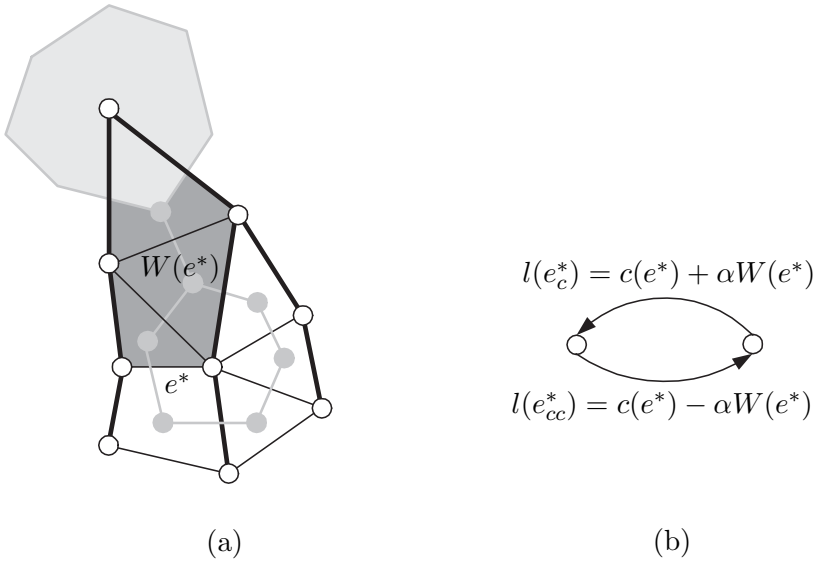
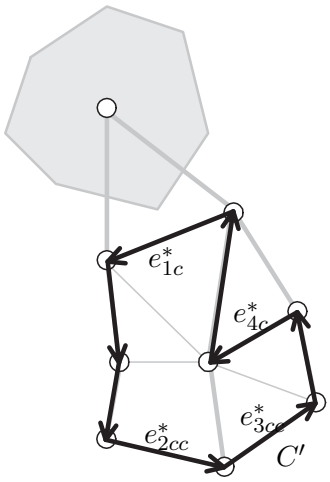


Figure 5.4: How the arcs e_c^* and e_{cc}^* are constructed for a non-tree edge e^*: (a) the darker shaded area shows the weight $W(e^*)$ enclosing by e^*, and (b) the lengths of e_c^* and e_{cc}^*.



$$\sum_{e^* \in C'} l(e^*) = c(C') - \alpha(W(e_2^*) + W(e_3^*) - W(e_4^*) - W(e_1^*))$$

Figure 5.5: The length of cycle C'.

We first argue that the second case is not possible, provided that all sparsest cuts cross F. To see this, note that the sparsity of S is at most the current ratio α, because the weights of nodes inside that cut is at most the demands crossing it; it is strictly less than the real demands when some source in the cut has its sink on the face between (e_i, e_j). Furthermore, S has a better ratio than any cut crossing F at (e_i, e_j), including ones which are the sparsest cuts; this leads to a contradiction.

We deal with the first case. If C^* crosses the face, it crosses F at (e_i^*, e_j^*) because of the way we modified the capacities on the other edges around F. Since it minimizes the ratio, which equals to the sparsity, it must be the sparsest cut.

It is left to consider the situation when C^* is clock-wise. In this case, the length of C^* is $c(C^*) + \alpha(\sum_{v \in C^*} w(v))$. The weight inside the cut is always non-negative when C^* crosses the face; thus, we can take the reverse of C^*, which is counter-clockwise and has no longer length. The case that C^* does not cross the face is similar to the second case above, and can be shown to lead to a contradiction using the same argument. ∎

By enumerating all possible pairs (e_i, e_j) we find the sparsest cut if it crosses the face F provided that all sparsest cuts cross the face.

We turn our attention to the case that none of the cuts crosses the face. To find the sparsest cut, we assign infinite capacities to all the boundary edges of F. For each node v not lying on F, we let $w(v)$ be the number of demands having v as their sources. We again have the problem of finding the minimum ratio cut, and it can be solved using the same idea as we presented earlier.

Thus, we have the following theorem.

Theorem 5.7 *Given a capacitated planar graph with a set of source-sink pairs where all the sinks lie on the same face, the sparsest cut can be found in polynomial time.*

Chapter 6

Minimum cuts in planar graphs

6.1 Overview

The *minimum cut problem* is to find the cut that minimizes the weighted sum of edges whose ends are separated by the cut. The size of the minimum cut is the weighted *edge-connectivity* of the graph. The algorithms for finding minimum cut [GH61, ET75, Tar74] were developed in the same line of research for max-flow algorithms [FF62], which find the minimum st-cut, the minimum cut that separate the source s and the sink t of the flow. Therefore, to find the minimum cut, one can run $O(n)$ max-flow computations. Podderyugin [Pod73], Karzanov and Timofeev [KT86], and Matula [Mat87] independently showed that minimum cuts in unweighted graphs can be found in time essentially comparable to one max-flow computation. Hao and Orlin [HO92] showed that the preflow-push framework of Goldberg and Tarjan [GT88] can be modified to find minimum cuts in weighted graphs. Other researchers [Gab91b, Gab91a, GW88, NI92] also introduced different techniques to find the minimum cuts in directed and undirected graphs. However, the running times of these algorithms were no better than the time needed to compute the minimum st-cut.

Karger and Stein [KS96] were the first to show that randomization can be used to get a substantial improvement in the running time, implying that finding global minimum cuts seems to be easier than finding the minimum st-cut. Their algorithm runs in $\tilde{O}(n^2)$ time. Karger [Kar00] improved this result to get a nearly linear-time $O(m \log^3 n)$ randomized algorithm. His algorithm works in general graphs. This is also the best previous running time for finding minimum cuts in planar graphs.

For planar graphs, we give a simple deterministic algorithm that runs in time $O(n \log^2 n)$

which exploits the structure of the minimum cuts and the shortest paths in the dual graphs. The algorithm is a divide-and-conquer one. In the dividing step we partition the graph along a shortest path tree in the dual and this allows us to find the minimum cut that crosses the partition efficiently. The algorithm that partitions the graph is similar to one of the subroutines in Lipton-Tarjan's separator algorithm [LT79] that, given a planar graph and a spanning tree, partitions the graph using some fundamental cycle determined by the tree and a non-tree edge. They use this subroutine on small-diameter spanning trees to get small node separators. We, however, use it on shortest path trees in the dual and have a different way of placing weights.

We note that in the case of unweighted planar graphs, the sizes of the minimum cuts are at most 5 since in any planar graph, there exists a node whose degree is at most 5. Thus, the problem of finding minimum cuts in unweighted graph reduces to the problem of determining if the dual graph contains cycles of fixed length where Alon, Yuster, and Zwick gave an $O(n)$-expected-time algorithm and an $O(n \log n)$-time algorithm [AYZ95].

6.2 The algorithm

Our algorithm is a divide-and-conquer one. The key ingredient for the algorithm is as follows. We use a shortest path tree in the dual graph to decompose the graph into two subgraphs; the structure of the minimum cuts and the shortest path tree in the dual graphs allows us to find the minimum cut which lies across the two subgraphs efficiently. Specifically, in that case, we can find a pair of nodes which are separated by the minimum cuts. Thus we need only a single application of the max-flow algorithm, which can be computed very efficiently using Weihe's algorithm [Wei97], to find the cut. We then recurse on the subgraphs, finding the minimum cut which lies entirely in the one of the subgraph. We return the best cut we find. Figure 6.1 illustrates the dividing step.

We describe the dividing step in Section 6.2.1 and the conquering step in Section 6.2.2.

6.2.1 Minimum cuts and a shortest path tree: the dividing step

Recall that a cut in the primal graph corresponds to a cycle in the dual. Therefore, minimum cuts corresponds to cycles of minimum length in the dual.

We say that paths p_1 and p_2 cross if they share some node. The number of times that p_1 and p_2 cross is number of paths (including paths with no edges) of $p_1 \cap p_2$. Using the

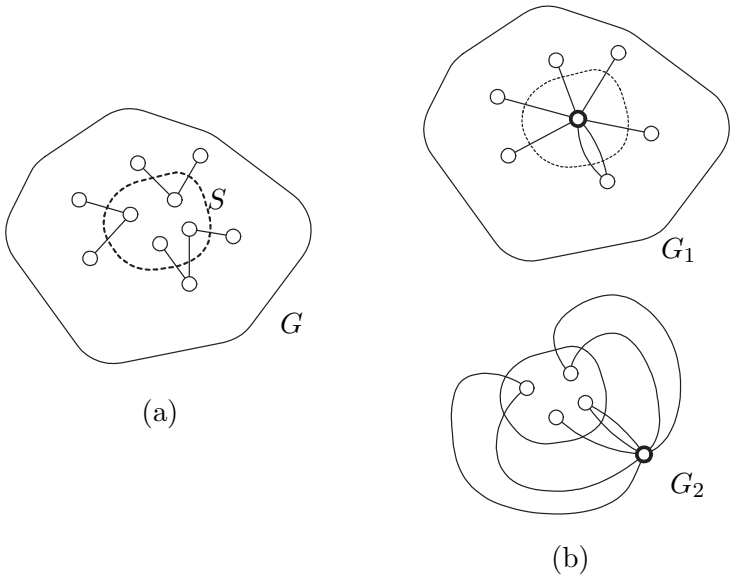


Figure 6.1: The dividing step: (a) G and the dividing cut S, and (b) two subproblems G_1 and G_2, each with a supernode shown as a thick ball to represent the rest of the graph.

duality, we can talk about a cut in terms of a cycle in the dual graph; therefore, we can defined the number of times a cut crosses any dual path similarly. We prove the following lemma stating that given a shortest path p in the dual graph, we only need to consider cuts which cross p at most once.

Lemma 6.1 *For any cut C in G which crosses any shortest path p in the dual G^* of G more than once, there exists a cut C' which crosses p at most once with no worse cost.*

Proof: Consider a cycle C^* in G^* corresponding to C. To prove the lemma, note that since C^* crosses p more than once, we can use p to short-cut C^* and get C'^*, which is no longer than C^*, with the corresponding cut C' as required. ∎

For any graph G, consider its dual G^*. Suppose that we have a shortest path tree T^* in G^* whose source is some node s^*. From Lemma 6.1, we know that there exists some minimum cut C that crosses any path from each leaf in T^* to s^* at most once. Now consider a cycle S^* in G^* with the following properties:

- there are three nodes in S^*, namely, a^*, b^*, and c^*, a common ancestor of a^* and b^* in T^*, which decompose S^* into 3 paths,

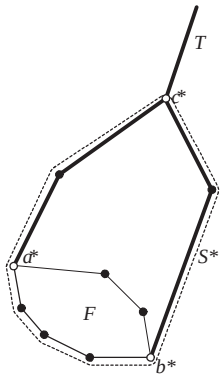


Figure 6.2: The dividing cut as a cycle S^* in G^*.

- the paths from a^* to c^* and from b^* to c^* are parts of the shortest path tree T^*, and
- the path from a^* to b^* is a boundary of some face F.

An example for S^* is shown in Figure 6.2. We will use a primal cut S corresponding to S^* to decompose the graph into two subgraphs.

To find a cut with these properties, we reduce the problem to the problem of finding edge separators in trees as shown in the following lemma.

Lemma 6.2 *Given any spanning tree $\mathcal{T}$ in a triangulated planar graph $G = (V, E)$, we can find, in linear time, the fundamental cycle determined by some non-tree edge e and $\mathcal{T}$ that separates the graph into two subgraphs, one inside the cycle and another outside, each containing at most $2/3$ fractions of the faces.*

Proof: Consider the dual $G^* = (V^*, E^*)$ of G. Let $\mathcal{T}^*$ be the set of dual edges corresponding to all the non-tree edges in $G - \mathcal{T}$. We first note the standard fact: $\mathcal{T}^*$ is a spanning tree in G^*. Now, note that edge e we want to find corresponds to some edge e^* in $\mathcal{T}^*$. The size of each subgraph separated by the cycle created by e and $\mathcal{T}$ is exactly the size of each subtree we get by deleting e^* from $\mathcal{T}^*$.

The problem now is to find an edge separator in $\mathcal{T}^*$. Since G is triangulated, each node in $\mathcal{T}^*$ has degree at most 3. It is known that one can find such an edge in linear time. ∎

This lemma is very similar to one in Lipton and Tarjan's separator algorithm [LT79] where they showed how to find a small node separator which is a fundamental cycle in a

low-diameter spanning tree. However, they put the weights on the nodes, while we put the weights on the faces.

In the algorithm, we have a shortest path tree T^* in the dual graph G^*. To apply the lemma, we triangulate G^*, thus splitting some number of nodes in G. For each edge we added, we set the cost to be infinity. We then use the lemma. If the non-tree edge $e^* \in G^*$ that we find is a triangulating edge on some face corresponding to some node v, we put v in the smaller subgraph.

6.2.2 Finding the minimum cut: the conquering step

We assume that we have the cut S and show how to find the minimum cut which crosses S using one max-flow computation. The main problem is to identify two nodes on different sides of the minimum cut.

Let P^*_{ac} denote the path in S^* from a^* to c^*. Also we define P^*_{bc} and P^*_{ab} similarly. Let v_F be the node in the primal graph corresponding to the face F whose boundary lies path P^*_{ab}. Since the cut forms a cycle, or a closed curve, in dual planar graph, it partition the graph into 2 subgraphs. We now call the subgraph which contains the node v_F as the *inside* of the cut. The other subgraph is called the *outside* of the cut. Because of this definition, we always use v_F as the source for the max-flow computation. The problem remains to find the node on the other side.

Let denote by e^*_a the last edge on P^*_{ac} adjacent to c^*. Let face F_a be the face outside S^* adjacent to e^*_a. We want to show that the minimum cuts cannot cross the cut S and contains both F and F_a. If this is true, to find the minimum cut, we only need to find the minimum $v_F v_{F_a}$-cut by a single max-flow computation, where v_{F_a} is a node in G corresponding to F_a. We prove this fact in the following lemma.

Lemma 6.3 *Any cycle C^* in G^* separating G^* into two subgraphs the outside and the inside which contains F and F_a must either*

- *cross the path P^*_{ac} or P^*_{bc} more than once, or*
- *not cross the cycle S^* at all.*

Proof: Assume that C^* crosses S^*; thus, it must cross S^* even number of times. It must cross either P^*_{ac} or P^*_{bc} more than once if it crosses S^* more than two times. Therefore, we

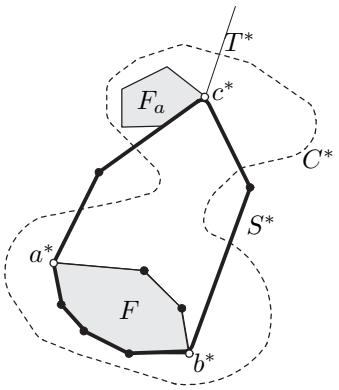


Figure 6.3: Any cut C^* that crosses S^* and contains both F and F_a must crosses either P^*_{ac} or P^*_{bc} more than once.

can assume that C^* only crosses S^* twice. This implies that C^* crosses P^*_{ac} once and P^*_{bc} once. It is clear that C^* cannot contain both F and F_a. Figure 6.3 depicts this proof. ∎

Therefore, to find the minimum cut crossing the dividing cut S, we run one max-flow computation from v_F to v_{F_a}. We use the max-flow algorithm in planar graphs by Weihe [Wei97] which runs in time $O(n \log n)$. This yields the following lemma.

Lemma 6.4 *The minimum cut that crosses the dividing cut S can be found in a single computation of max-flow; thus, in $O(n \log n)$ time, we can find such cut.*

The main result follows.

Theorem 6.5 *The minimum cut in a planar graph can be found in $O(n \log^2 n)$ time.*

Proof: We find the dividing cut S in linear time. We recursively find the minimum cuts in both subproblems. Then, in $O(n \log n)$ time, we find the minimum cut that crosses S. We return the minimum. The running time, denoted by $T(n)$, is $T(n_1)+T(n-n_1)+O(n \log n)$, where $n/3 \leq n_1 \leq 2n/3$; therefore, $T(n) = O(n \log^2 n)$. ∎

Chapter 7

Graph decomposition

7.1 Overview

Graph decomposition finds applications in many network problems. The general aim is to produce a partition which minimizes the number of edges between difference component of the graph while keeping the radius of each component small.

For general graphs, it is known that it suffices to cut an $O(\log n/\delta)$ fraction of the edges to decompose it into clusters of diameter δ, and this is the best one can do for general graphs.

Klein, Plotkin, and Rao [KPR93] considered restricted families of graphs, and showed that given a parameter δ, one can decompose a graph excluding a $K_{r,r}$ minor into clusters of diameter δ, while cutting only $O(r^3/\delta)$ fraction of the edges.

Note that any such decomposition of a path graph must cut an $O(\frac{1}{\delta})$ fraction of the edges, and thus the overhead for graphs excluding $K_{r,r}$ was shown to be $O(r^3)$. Not surprisingly, this decomposition lemma has found several other applications to approximation algorithms, distributed computing and embeddings results for such graphs.

In this paper we make some progress towards finding the right relation between the size of the forbidden minor and the overhead of such a decomposition. We show that for any graph excluding a K_r minor, we can find a decomposition into clusters of diameter δ while cutting only $O(r^2/\delta)$ fraction of the edges. This improves the performance guarantees of approximation algorithms and embeddings results for such graphs.

7.1.1 The techniques

The techniques used in this paper borrow very generously from those used by Klein, Plotkin, and Rao [KPR93]. They showed that if their algorithm of repeatedly shattering breadth-first search trees $O(r)$ times produced a cluster of large diameter, then they could construct a $K_{r,r}$ minor, consisting of r well spaced points in the large diameter cluster and the r roots of the breadth-first search trees. We note that the roots of the breadth-first search trees used were chosen arbitrarily.

Instead, we are somewhat more careful in our choice of the roots. We make sure that the roots of the breadth-first search trees constructed are mutually far apart; this allows us to construct disjoint paths connecting these roots. Instead of r well spaced points in a high diameter cluster, we only use two, along with the roots of the breadth-first search trees to construct a K_r minor. This allows us to get a better guarantee on the diameter of the clusters.

7.1.2 Applications

Klein, Plotkin and Rao [KPR93] developed this graph decomposition technique for graphs excluding a fixed size minor in order to prove the approximate max-flow min-cut theorem for this class of graphs. In particular, they showed that for planar graphs, which exclude K_5 and $K_{3,3}$ minors, the max-flow min-cut gap is $O(1)$ for the uniform case.

The decomposition theorem itself has found applications to approximation algorithms for various NP-hard problems. We mention a few of these applications here. Rao and Richa [RR98] gave $O(r^3 \log\log n)$-approximation algorithms for minimum linear arrangement and minimum containing interval graph on graphs excluding K_r minor. Calinescu, Karloff and Rabani [CKR01] gave an $O(r^3)$-approximation algorithm for the 0-extension problem on such graphs and Feige and Krauthgamer [FK02] gave an $O(r^3 \log n)$-approximation algorithm to minimum bisection on such graphs.

A slight modification of these decompositions have also been used in the area of metric embeddings. Rao [Rao99] showed that graphs excluding K_r minors can be embedded into l_2 with distortion $O(r^3\sqrt{\log n})$. Moreover these embeddings preserve not only distances but also volumes. Recently, Rabinovich [Rab03] showed how to embed a metric excluding K_r into a line with *average distortion* $O(r^3)$. For graphs with tree width r, they further improved the embedding to $O(\log r)$ and left open the question of the correct order for

graphs excluding K_r minor. Our results improve the r^3 in all the above applications to r^2.

7.2 Preliminaries

Let H and G be graphs. Suppose that for every vertex v of H, G contains a connected subgraph $\mathcal{A}(v)$ and for every edge (u, v) in H, there is an edge $\mathcal{E}(uv)$ connecting $\mathcal{A}(u)$ and $\mathcal{A}(v)$ in G. If the $\mathcal{A}(v)$'s are pairwise disjoint, we say that G contains an *H-minor* and call $\cup_v \mathcal{A}(v)$ *an H-minor* of G. We refer to the $\mathcal{A}(v)$'s as *supernodes* and $\mathcal{E}(uv)$'s as *superedges*.

We denote by K_h the complete graph on h nodes. Note that if G contains a K_h minor, it contains every minor on h vertices. Thus if G excludes any minor of size h, it excludes K_h.

Given a graph $G = (V, E)$, we can define a natural distance measure on V - $d_G(u, v)$ is the length of the shortest path from u to v. For a subset V' of V, the *weak* diameter of V' is defined to be $\max_{u,v \in V'}\{d_G(u, v)\}$. The induced subgraph G' on V' defines a metric $d_{G'}$. The *strong* diameter of V' is the diameter under the metric $d_{G'}$.

A δ-decomposition of $G = (V, E)$ is a partition of V into subsets $V_1, V_2, \ldots, V_k$ such that each cluster V_i has (weak) diameter at most δ. An edge $e = (u, v)$ is said to be cut by this decomposition if u and v lie in different V_i's.

7.3 The decomposition procedure

We decompose the graph recursively $r - 2$ times. At each level i, given a cluster G_i, we do the following. We pick, if possible, an *appropriate* node (explained in the next paragraph) a_i in G_i and construct a breadth first search tree rooted at a_i. We say a vertex v is at *level l* if its distance in G_i, from a_i is l. We partition the edges of G_i into δ classes. For $k = 0, 1, \ldots, \delta - 1$, the k^{th} class consists of edges between nodes at level $i\delta + k$ and $i\delta + k + 1$ for some integer $i \geq 0$. Since each edge belongs to at most one class and we have δ classes, there exists one class of edges which contains no more than $E(C)/\delta$ edges. We cut these edges and recurse on the resulting components.

By *appropriate* above, we mean a node which is at least distance $4r\delta$ far from each of roots of the breadth-first search trees in the higher levels of recursion. In case there is no such node in cluster G_i, we shatter the cluster in a different way - each cluster consisting of vertices close to one of the previous level roots.

We then further shatter each resulting cluster G_{r-1} into at most $r-1$ pieces by cutting out clusters of inappropriate nodes. For each of the centers $a_1, \ldots, a_{r-2}$, we delete a set of vertices close to a_i. We redefine G_{r-1} to be the remaining set of nodes C'.

Figure 7.1 show the pseudocode of the procedures. We start by calling Decompose($G_1 = G, 1, \{\}$).

7.4 Proof of the decomposition procedure

We follow closely the approach of Klein, Plotkin and Rao [KPR93]. Given a graph G_1 and a distance parameter δ, they consider a series of at most $r+1$ subgraphs $G_2, G_3, \ldots, G_{r+1}$ of G_1. Each graph G_{i+1} is a connected component of a subgraph of G_i induced by δ consecutive levels of a breadth-first search tree $\mathcal{T}_i$ starting from an arbitrary node a_i in G_i. If on the last subgraph G_{r+1} of the series, the distance between some pair of nodes is larger than $O(r^2\delta)$, they showed that G_1 contains $K_{r,r}$ as a minor.

We try to construct a K_r-minor instead. Given a connected graph G_1 and a distance parameter δ, we construct a series of at most $r-1$ subgraphs $G_2, G_3, \ldots, G_{r-1}$ of G_1. Our breadth-first search trees, however, do not start from arbitrary nodes. For $\mathcal{T}_i$, we select an appropriate root node a_i such that for $j < i$, the distance from a_i to a_j in G_1 is at least $4r\delta$. If we cannot find such a node, the recursion terminates.

We first show that in the latter case, we can further decompose G_i as in procedure Shatter such that we cut at most $O(r|E(G_i)|/\delta)$ edges and each resulting component has weak diameter $O(r\delta)$.

Lemma 7.1 *If the decomposition procedure stops at G_i because there is no appropriate root, procedure* Shatter *finds at most $r|E(G_i)|/\delta$ edges whose removal decompose G_i into components each of weak diameter at most $(8r+2)\delta$.*

Proof: For each $j = 1, \ldots, i-1$, we define the set C_j to be the set of all vertices in G_i which are at distance at most $4r\delta$ from a_j. Create a breadth-first tree from set C_1, and consider only the first $\delta + 1$ levels. We can define a sequence of cuts $S_1, \ldots, S_\delta$ by considering edges spanning different levels of the tree. One of them contains at most $|E(G_i)|/\delta$ edges and we cut at such a level. We then repeat for each of $C_2, C_3, \ldots, C_{i-1}$.

In the process, we have cut at most $(i-1)|E(G_i)|/\delta \leq r|E(G_i)|/\delta$ edges. We now show that the diameter of each component is at most $(8r+2)\delta$. Consider any pair of nodes u

Algorithm Decompose($G_i, i, p = \{a_1, \ldots, a_{i-1}\}$)

1. **if** there exists $r \in G_i$ such that $d_G(a_j, v) \geq 4r\delta$ for all $1 \leq j \leq i-1$ **then**
1.1 $a_i \leftarrow r$.
1.2 Create a breadth-first search tree $\mathcal{T}_i$ in G_i rooted at a_i.
1.3 **if** $\mathcal{T}_i$ contains less than $\delta + 1$ level **then**
1.3.1 **stop**.
1.4 **for** $k = 0, 1, \ldots, \delta - 1$ **do**
1.4.1 Define the k-th cut S^k to be the set of edges between nodes at level $j\delta + k$ and $j\delta + k + 1$ in $\mathcal{T}_i$, for some $j \geq 0$.
1.5 Let S be a cut S^k that has at most $|E(C)|/\delta$ edges.
1.6 Cut all edges in S.
1.7 **for** each component G' in $G_i - S$ **do**
1.7.1 **if** $i < r - 2$ **then**
1.7.1.1 Decompose($G', i + 1, \{a_1, \ldots, a_{i-1}, a_i\}$).
1.7.2 **else**
1.7.2.1 Shatter($G', i, \{a_1, \ldots, a_{i-1}, a_i\}$).
2. **else**
2.1 Shatter($G_i, i - 1, p$).

Procedure Shatter($C, k, p = \{a_1, \ldots, a_k\}$)

1. $C' \leftarrow C$.
2. **for** $j = 1, \ldots, k$ **do**
2.1 $C_i \leftarrow$ all nodes v in C' such that $d_G(v, a_i) \leq 4r\delta$.
2.2 Create a breadth-first search tree T_i from nodes in C_i.
2.3 Let T_i' be the first $\delta + 1$ levels of T_i.
2.4 **if** T_i' covers all C' **then**
2.4.1 $C' \leftarrow \emptyset$.
2.5 **else**
2.5.1 Let j be such that at most $E(C)/\delta$ edges go from level j to level $(j + 1)$.
2.5.2 Let T_i'' be a subtree of T_i up to level j.
2.5.2 Cut all edges at level j.
2.5.3 $C' \leftarrow C' - (C_i \cup T_i'')$.

Figure 7.1: The decomposition procedures.

and v in the same connected component T_i'' in the resulting graph. It must be the case that there is some a_i such that the distance from u and v to a_i is at most $(4r+1)\delta$ in G_1. Therefore, the weak diameter of each such component is at most $(8r+2)\delta$.

Finally, note that since there was no appropriate vertex in C, the C_j's defined above cover all of C, and hence C' is empty at the end of the procedure Shatter. ∎

Therefore, if the series stops before $i = r-1$ (either because the diameter of the cluster was small, or the cluster had no appropriate node), our decomposition has the required property. Moreover, note that the above proof also shows that when Shatter is called in line 1.7.2.1 of the algorithm, weak diameter of every cluster T_i'' cut out by procedure Shatter is small; only the remaining set of vertices G_{r-1} can have large diameter.

We now consider this case. We wish to show that if the graph excludes a K_r minor, then the diameter of each such cluster resulting from our decomposition algorithm is small. We shall show the contrapositive - if the resulting decomposition has some cluster with large diameter, we shall show how to construct a K_r minor in the graph. Let G_{r-1} be a cluster of large diameter and let a_{r-1} and a_r be two vertices in G_{r-1} which are at least distance $4r\delta$ apart. We shall construct a K_r minor, containing a supernode centered at each a_i, for $i = 1, 2, \ldots, r$. We shall use the paths in the breadth-first search trees to find superedges.

Lemma 7.2 *Suppose that a cluster G_{r-1} output by our algorithm has diameter $4r\delta$. Then G_1 contains a K_r minor.*

Proof: As above, denote by a_{r-1} and a_r two nodes in G_{r-1} at distance $4r\delta$ from each other. Note that by our construction, every pair of a_i and a_j is at least distance $4r\delta$ apart.

We shall show how to construct a K_r minor in G_1. We do so by reverse induction - we give a procedure which, for $b = r-2, r-3, \ldots, 1$, constructs a K_{r-b}-minor in G_{b+1}.

Recall that G_{i+1} consists of δ consecutive layers in the breadth-first search tree $\mathcal{T}_i$ rooted at a_i. An *ancestor-path* of v in $\mathcal{T}_i$ is the path in $\mathcal{T}_i$ from v to the root a_i of $\mathcal{T}_i$. We shall construct the minor using suitable ancestor-paths in $\mathcal{T}_i$'s.

Given a K_b-minor in G such that starting at each supernode $\mathcal{A}(g)$ there is path P_g, we say that the paths $\{P_g\}$ are *tails* if each path P_g is disjoint from the other paths and also from all supernodes except $\mathcal{A}(g)$. We shall refer to P_g's ending node (outside of $\mathcal{A}g$) as the *tip* of the tail P_g and denote it by $tip(P_g)$.

Klein, Plotkin and Rao [KPR93] show how to construct the minor inductively by also constructing tails which are ancestor-paths of $\mathcal{T}_b$ and special nodes (which they called

middle nodes) on the tails which are far apart, and use them to further construct disjoint components of the minor. We use a similar approach.

We shall construct a K_{r-b}-minor in G_{b+1}. In addition, we construct $r-b+1$ tails $\{P_i\}$ which are ancestor-paths of $\mathcal{T}_b$ of length exactly 4δ such that for each tail P_i, a middle node h_i of P_i is at distance $4b\delta$ from the other middle nodes h_j's. Moreover, we require that every middle node is at distance at least $4b\delta$ from the root a_b of $\mathcal{T}_b$. This shall be our (reverse) inductive claim.

For the basis step, when $b = r-2$, let P be the shortest path from a_{r-1} to a_r in G_{r-1}. We construct a K_2-minor from the path P. We let $\mathcal{A}(a_r)$ be a path of length $4\delta - 1$ on P starting from a_r. The other supernode $\mathcal{A}(a_{r-1})$ is then $P - \mathcal{A}(a_r)$. We construct the tails by taking P_j to be the ancestor-paths in $\mathcal{T}_{r-2}$ of length 4δ from a_j, for $j \in \{r, r-1\}$. It can be checked that they are proper tails and the middle nodes in these tails are at distance at least $4(r-2)\delta$ from each other. Also the middle nodes in these tails are at distance at least $4(r-2)\delta$ from a_{r-2}.

We now show the inductive step. Assuming that the claim is true for $b = i+1$, we want to show that the claim is true for $b = i$, i.e., G_{i+1} contains K_{r-i} as a minor and a new set of tails with the required properties.

We first construct the minor. For $j > i+1$, we create supernodes $\mathcal{A}'(a_j)$ from the supernodes of K_{r-i-1} as follows. We let $\mathcal{A}'(a_j)$ be $\mathcal{A}(a_j) \cup (P_j - \{tip(P_j)\})$. From the inductive assumption, these supernodes are disjoint. This gives us $r-i-1$ supernodes. We let $\mathcal{A}'(a_{i+1})$ be a union of all ancestor-paths in $\mathcal{T}_{i+1}$ starting from the tip of all the tails $\{P_j\}$.

We must show that $\mathcal{A}'(a_{i+1})$ is disjoint from all other new supernodes. Since we create tails of length 4δ from the ancestor-paths in $\mathcal{T}_{i+2}$, the end nodes of the tails lie outside the subgraph G_{i+2}; therefore, the supernodes $\mathcal{A}(a_j)$, lying inside G_{i+2}, and $\mathcal{A}'(a_{i+1})$ are disjoint. Also, the tails $\{P_j\}$ and $\mathcal{A}'(a_{i+1})$ are disjoint by construction. Moreover, the last edges on the paths P_j give us the required additional superedges. This shows that G_{i+1} contains a K_{r-i}-minor.

To finish the inductive claim, we need to construct the tails with the desired properties. For each middle node h_j, let the tail P'_j be the ancestor-paths in $\mathcal{T}_i$ from h_j of length 4δ. These tails are mutually disjoint because h_j's are at distance at least $4(i+1)\delta$ from each other in G_1, hence also in G_b. We also create another tail P'_i starting from a_i in the same

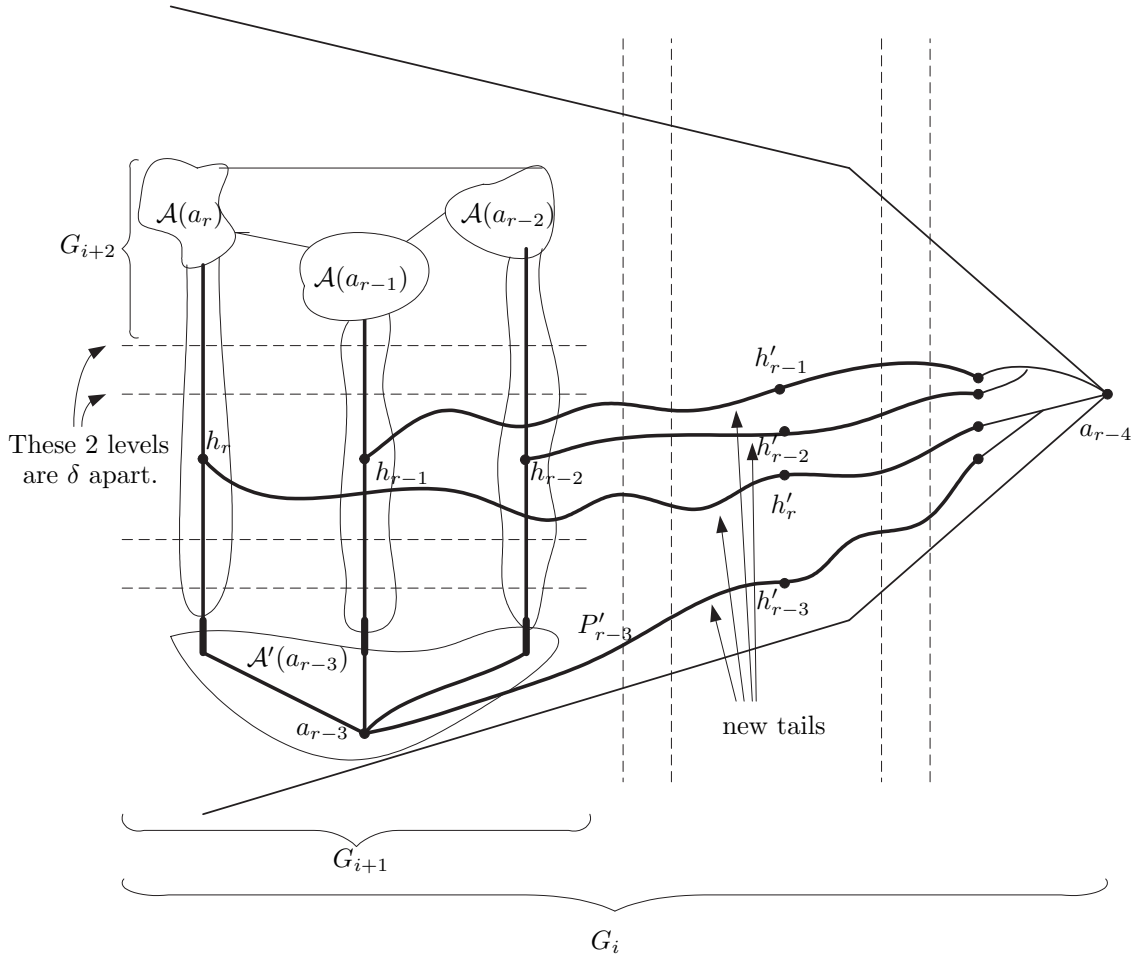


Figure 7.2: Inductive step.

way. It is straightforward to verify that the new middle nodes $\{h'_j\}$ are at the right distance of each other.

We must also show that the tail P'_j are disjoint from all $\mathcal{A}'(a_k)$ where $k \neq j$. Consider any node v in $\mathcal{A}'(a_k)$. From the choice of h_j, the levels of v and h_j in $\mathcal{T}_{i+1}$ differ by more than δ. This implies that v does not lie on the ancestor-paths of h_j in $\mathcal{T}_i$[1] for any j. Thus for any $j \neq k$, P'_j is disjoint from $\mathcal{A}'(a_k)$. To show that P'_j does not cross any tails P_k, we note that the distance between h_j and h_k is more than 6δ. Finally, since a_i is at distance $4(i+1)\delta$ from all the middle nodes h_j, the path P'_i is also a proper tail.

It only remains to show that the middles node h'_j's are at distance at least $4i\delta$ from a_{i-1}. From our construction a_{i-1} is at distance at least $4r\delta$ from a_j, where $j > i$. We know inductively that the new middle node h'_j are at distance at most $2(r-i)\delta$ from a_j. By triangle inequality then, the distance from h'_j and a_{i-1} is at least $4r\delta - 2(r-i)\delta \geq 4i\delta$. This completes the inductive argument.

Thus, when $b = 1$, the induction claim says that G_2 contains a K_{r-1}-minor and the tails with the appropriate properties. We can construct a K_r-minor in G_1 as in the inductive step. This completes the proof of Lemma 7.2. ∎

From the above two lemmas, we have the main theorem.

Theorem 7.3 *Given a graph G and parameters δ and r, we can either find a K_r minor in G or find a decomposition of the G which cuts at most $O(mr/\delta)$ edges such that the weak diameter of each piece of the decomposition is at most $O(r\delta)$.*

Proof: For each of the level decomposition, procedure Decompose cuts at most m/δ edges. There are at most $r-2$ levels; thus, at most $m(r-2)/\delta$ edges are cut in Decompose. Procedure Shatter cuts at most mr/δ edges. Therefore, we cut at most $2m(r-1)/\delta$ edges.

From the lemmas above, we either find the K_r minor or the decomposition constructed has weak diameter at most $(8r+2)\delta$. ∎

This result can be extended to work with graphs which exclude $K_{r,r}$ minor as well, because K_{2r} contains $K_{r,r}$ as a minor. We can also generalize this procedure for graphs with distances and weights on the edges. Moreover, this algorithm can be easily "randomized" so that for each edge, the probability that it gets cut is $O(r/\delta)$ and we get clusters of diameter $O(r\delta)$.

[1]This is exactly the "moat" argument in [KPR93].

Chapter 8

Conclusions and open problems

This dissertation presents algorithms for some path and flow problems in planar graphs. The main ingredient, except for the last result on graph decompositions, is the topological structure of shortest paths which provides information on the solutions we look for.

We conclude by listing various open problems grouped by topics.

Shortest paths

We start with some improvement on the algorithms we present. Recall that we handle the general Monge searching problem by using $O(\log n)$ bipartite Monge arrays.

Open problem 1. Is it possible to handle the general Monge array directly? This might improve the running time by a factor of $O(\log n)$.

The techniques for exploiting the non-crossing property that we use here depends severely on the arrangement of nodes. This does not work in many cases where the nodes lie arbitrarily, e.g., when they are points on the plane (see, e.g., the bipartite matching on the plane [Vai89, AES99, VA99] and the minimum fuel-consumption problem [CE01]).

Open problem 2. Is there a way to use the non-crossing property in other settings?

With this shortest path algorithm we can implement Miller-Naor's algorithm [MN95] which finds the flows when there are multiple sources and multiple sinks efficiently. However, for some application in computer vision (see [CRZ96]), the problem that we want to test for feasibility contains only one source with multiple sinks. Note that Miller-Naor algorithm

essentially finds negative cycles in the dual graphs. In this case, the structure of negative cycles is quite simple. By using a shortest path tree in the dual with the same argument as in [Rao92], we can show that at least one of the cycles lie "nicely," i.e., it crosses any shortest path on the tree at most twice. With this much structural property of the negative cycles, we still do not know how to find one of them.

Open problem 3. Can one do better than Miller-Naor's algorithm when there is only one source? (We hope for a linear time algorithm here.)

Miller-Naor's algorithm only tests for flow feasibility.

Open problem 4. Can one efficiently find maximum flows in planar graphs with multiple sources and sinks?

The last one is a dream.

Open problem 5. Improve Bellman-Ford.

Multicommodity flows

It might be possible that the two-step routing that we use is applicable to other kinds of graphs.

Open problem 6. Is there a way to extend this technique to larger classes of planar graphs?

The next one is a recent classic.

Open problem 7. Find the right approximate max-flow min-cut gap for planar graphs.

The same relation for the fractional flows and integral flows is very interesting. The best upperbound for general graphs is $O(\log n / \log\log n)$ using randomized rounding [RT87]. It has been proved to be constant for densely embeddable planar graphs by [KT95] (see also in [Kle96]). Leighton gave an $\Omega(\log n / \log\log n)$ example for directed planar graphs, thus, settling the problem in the directed case. But nothing is known for undirected planar graphs, where the worst examples show the gap of 2. If one can show a rounding procedure,

this would give also an approximation algorithm that minimizes congestion as well. We formulate the problem as follows.

Open problem 8. Given a fractional solution of the flow problems in undirected planar graphs, show that the integral one with small congestion exists.

This problem is the same as the conjecture by Seymour [Sey81] for general graphs which state that the above gap is 2. It was disproved by Lomonosov [Lom85].

Minimum cuts

We focus on the problem in general graphs.

Open problem 9. The recent nearly linear-time algorithm of Karger [Kar00] needs one tree which crosses the minimum cut at most twice. If one can find such a tree quickly, one can derandomize Karger's algorithm.

Satish Rao noted that the time to find the minimum cut that crosses a given spanning tree once is much better than to find the cut that crosses the tree twice. He asked the following question.

Open problem 10. Can this fact be used? E.g., can one find a tree packing with the following properties: either one of the trees crosses the minimum cut only once, or one can find, among the trees in the packing, a small number of trees, one of which crosses the cut twice?

Graph decompositions

Rabinovich [Rab03] asked the following question.

Open problem 11. What is the right dependency on r to decompose graphs excluding a K_r as a minor?

Bibliography

[ABNK+95] Alok Aggarwal, Amotz Bar-Noy, Samir Khuller, Dina Kravets, and Baruch Schieber. Efficient Minimum Cost Matching and Transportation Using the Quadrangle Inequality. *Journal of Algorithms*, 19(1):116–143, 1995.

[AES99] P. K. Agarwal, A. Efrat, and M. Sharir. Vertical decomposition of shallow levels in 3-dimentional arrangements and its applications. *SIAM J. Comput.*, 29:912–953, 1999.

[AGLR94] Baruch Awerbuch, Rainer Gawlick, Frank Thomson Leighton, and Yuval Rabani. On-line admission control and circuit routing for high performance computing and communication. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 412–423, 1994.

[AH77a] K. Appel and W. Haken. Every planar map is four colorable. part i. discharging. *Illinois J. Math.*, 21:429–490, 1977.

[AH77b] K. Appel and W. Haken. Every planar map is four colorable. part ii. reducibility. *Illinois J. Math.*, 21:491–567, 1977.

[AP90] B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the Thirty-First Annual IEEE Symposium on Foundations of Computer Science*, pages 503–513, 1990.

[AR95] Yonatan Aumann and Yuval Rabani. Improved bounds for all optical routing. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 567–576. ACM Press, 1995.

[AR98] Yonatan Aumann and Yuval Rabani. An $O(\log k)$ approximate min-cut max-

flow theorem and approximation algorithm. *SIAM J. Comput.*, 27(1):291–301, 1998.

[AYZ95] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

[Bel58] R. E. Bellman. On a Routing Problem. *Quart. Appl. Math*, 16:87–90, 1958.

[Bol98] Béla Bollobás. *Modern Graph Theory*. Springer, 1998.

[BY98] Samuel R. Buss and Peter N. Yianilos. Linear and $O(n \log n)$ Time Minimum-Cost Matching Algorithms for Quasi-Convex Tours. *SIAM Journal on Computing*, 27(1):170–201, 1998.

[CE01] Timothy M. Chan and Alon Efrat. Fly cheaply: On the minimum fuel-consumption problem. *J. Algorithm*, 41:330–337, 2001.

[CGN+03] Chandra Chekuri, Anupam Gupta, Ilan Newman, Yuri Rabinovich, and Alistair Sinclair. Embedding k-outerplanar graphs into ℓ_1. In *SODA*, 2003.

[CKR01] Gruia Calinescu, Howard J. Karloff, and Yuval Rabani. Approximation algorithms for the 0-extension problem. In *Symposium on Discrete Algorithms*, pages 8–16, 2001.

[CRZ96] I. J. Cox, S. B. Rao, and Y. Zhong. 'Ratio Regions': A Technique for Image Segmentation. In *Proceedings International Conference on Pattern Recognition*, pages 557–564. IEEE, Aug. 1996.

[DGV95] L.A. Costa D. Geiger, A. Gupta and J. Vlontzos. Dynamic programming for detecting, tracking and matching elastic contours. *IEEE Trans. On Pattern Analysis and Machine Intelligence*, 1995.

[Die00] Reinhard Diestel. *Graph Theory*. Springer, second edition edition, 2000.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Dji82] H. N. Djidjev. A linear algorithm for partitioning graphs. *C. R. Acad. Bulgare Sci.*, 35(8):1053–1056, 1982.

[DPZ91] Hristo N. Djidjev, Grammati E. Pantziou, and Christos D. Zaroliagis. Computing Shortest Paths and Distances in Planar Graphs. In *Proc. 18th ICALP*, pages 327–339. Springer-Verlag, 1991.

[ET75] Shimon Even and Robert Endre Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975.

[FF62] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.

[FHW80] S. Fortune, J. E. Hopcroft, and J. Wylie. The directed subgraph homeomorphism problem. *Theor. Comput. Sci.*, 10:111–121, 1980.

[FK02] Uriel Feige and Robert Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM Journal on Computing*, 31(4):1090–1118, August 2002.

[FR01] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science*, pages 232–241, 2001.

[Fra90] A. Frank. Packing paths, circuits, and cuts - a survey. In B. Korte, L. Lovász, H-J. Prömel, and A. Schrijver, editors, *Paths, Flows and VLSI-Layouts*, pages 47–100. Springer Verlag, 1990.

[Fre87] Greg N. Frederickson. A new approach to all pairs shortest paths in planar graphs (extended abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 19–28, May 1987.

[Fre89] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, December 1989.

[FT03] Jittat Fakcharoenphol and Kunal Talwar. Improved decompositions of graphs with forbidden minors. manuscript, submitted, 2003.

[Gab91a] Harold N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proceedings of the 32th Annual Symposium on Foundations of Computer Science*, pages 812–821, 1991.

[Gab91b] Harold N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 112–122. ACM Press, 1991.

[GH61] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society of Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[GK98] Naveen Garg and Jochen Konemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *IEEE Symposium on Foundations of Computer Science*, pages 300–309, 1998.

[GNRS99] A. Gupta, I. Newman, Y. Rabinovich, and A. Sinclair. Cuts, trees and l_1-embeddings of graphs. In *40th FOCS*, pages 399–408, 1999.

[Gol92] Andrew V. Goldberg. Scaling algorithms for the shortest path problem. *SIAM Journal on Computing*, 21(1):140–150, 1992.

[Goo95] Michael T. Goodrich. Planar separators and parallel polygon triangulation. *Journal on Computer and System Sciences*, 51(3):374–389, 1995.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

[GT89] Harold N. Gabow and Robert E. Tarjan. Faster Scaling Algorithm for Network Problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[GVY97] Naveen Garg, Vijay V. Vazirani, and Mihalis Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1):3–20, 1997.

[GW88] Harold Gabow and Herbert Westermann. Forests, frames, and games: algorithms for matroid sums and applications. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 407–421. ACM Press, 1988.

[Har71] Frank Harary. *Graph Theory.* Addison-Wesley, 1971.

[Has81] Rafael Hassin. Maximum flow in (s, t) planar networks. *Information Processing Letters*, 13(3):107, 1981.

[HKRS97] Monika R. Henzinger, Philip N. Klein, Satish Rao, and Sairam Subramanian. Faster Shortest-Path Algorithms for Planar Graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

[HO92] Jianxiu Hao and James B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 165–174. ACM Press, 1992.

[HT74] John E. Hopcroft and Robert Endre Tarjan. Efficient planarity testing. *JACM*, 21(4):549–568, 1974.

[Joh77] D.B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. Assoc. Comput. Mach.*, 24:1–13, 1977.

[Kah93] N. Kahale. On reducing the cut ratio to the multicut problem. Technical Report 93-78, DIMACS, 1993.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.

[Kar00] David R. Karger. Minimum cuts in near-linear time. *Journal of the ACM (JACM)*, 47(1):46–76, 2000.

[KK03] Łukasz Kowalik and Maciej Kurowski. Short path queries in planar graphs in constant time. In *STOC'03 (to appear)*, 2003.

[Kle96] Jon M. Kleinberg. *Approximation Algorithms for Disjoint Paths Problems*. PhD thesis, Dept. of EECS, MIT, 1996.

[KP95] Anil Kamath and Omri Palmon. Improved interior point algorithms for exact and approximate solution of multicommodity flow problems. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 502–511. ACM Press, 1995.

[KPR93] Philip Klein, Serge A. Plotkin, and Satish Rao. Excluded minors, network decomposition, and multicommodity flow. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 682–690. ACM Press, 1993.

[KRAR95] Philip N. Klein, Satish Rao, Ajit Agrawal, and R. Ravi. An approximate max-flow min-cut relation for unidirected multicommodity flow, with applications. *Combinatorica*, 15(2):187–202, 1995.

[KS96] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, 1996.

[KT86] A. V. Karzanov and E. A. Timofeev. Efficient algorithm for finding all minimal edge cuts of a non-oriented graph. *Cybernetics*, 22:156–162, 1986.

[KT95] Jon M. Kleinberg and Eva Tardos. Disjoint paths in densely embedded graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 52–61, 1995.

[Kur30] Kasimir Kuratowski. Sue le problème des courbes gauches en topologie. *Fund. Math.*, 15:217–283, 1930.

[KvL84] M. E. Kramer and J. van Leeuwen. The complexity of wire routing and finding the minimum area layouts for arbitrary vlsi circuits. In F. P. Preparata, editor, *Advances in Computing Research 2: VLSI Theory*, pages 129–146, London, 1984. JAI Press.

[Lei80] C. Leiserson. Area-efficient graph layouts (for vlsi). In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 270–281, 1980.

[LLR95] Linial, London, and Rabinovich. The geometry of graphs and some of its algorithmic applications. *COMBINAT: Combinatorica*, 15, 1995.

[Lom85] M. Lomonosov. Combinatorial approaches to multiflow problems. *Discrete Appl. Math.*, 11:1–94, 1985.

[LR88] F. T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 256–269, 1988.

[LR99] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)*, 46(6):787–832, 1999.

[LRT79] R. Lipton, D. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16:346–358, 1979.

[LT79] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:177–189, 1979.

[LT80] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9(3):615–627, 1980.

[LY78] C. L. Lucchesi and D. H. Younger. A minimax theorem for directed graphs. *J. London Math. Soc.*, 17(2):369–374, 1978.

[Lyn75] J. F. Lynch. The equivalence of theorem proving and the interconnection problem. *(ACM) SIGDA Newsletter*, 5(3):31–36, 1975.

[Mat87] David W. Matula. Determining edge connectivity in $O(nm)$. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 249–251, 1987.

[Mil86] Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, 1986.

[MMP87] Joseph S. B. Mitchell, David M. Mount, and Christos H. Papadimitriou. The discrete geodesic problem. *SIAM Journal on Computing*, 16(4):647–668, 1987.

[MN95] G. Miller and J. Naor. Flow in planar graphs with multiple sources and sinks. *SIAM Journal on Computing*, 24:1002–1017, 1995.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge connectivity in multigraphs and capacitated graphs. *SIAM Journal of Discrete Mathematics*, 5(1):54–66, 1992.

[OS81] H. Okamura and P.D. Seymour. Multicommodity flows in planar graphs. *Journal of Combinatorial Theory, Series B*, 31:75–81, 1981.

[Pod73] V. D. Podderyugin. An algorithm for finding the edge connectivity of graphs. *Vopr. Kibern.*, 2:136, 1973.

[PP93] J. K. Park and C. A. Phillips. Finding minimum-quotient cuts in planar graphs. In *Proceedings of the 25th Annual ACM Symposium on the Theory of Computing*, pages 766–775, 1993.

[PST91] Serge A. Plotkin, David B. Shmoys, and Eva Tardos. Fast approximation algorithms for fractional packing and covering problems. In *IEEE Symposium on Foundations of Computer Science*, pages 495–504, 1991.

[PT93] Serge A. Plotkin and Eva Tardos. Improved bounds on the max-flow min-cut ratio for multicommodity flows. In *ACM Symposium on Theory of Computing*, pages 691–697, 1993.

[Rab03] Yuri Rabinovich. On average distortion of embedding metrics into l_1 and into the line yuri rabinovich. In *STOC*, 2003. To appear.

[Rao87] Satish B. Rao. Finding near optimal separators in planar graphs. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 225–237, 1987.

[Rao92] Satish B. Rao. Faster algorithms for finding small edge cuts in planar graphs (extended abstract). In *In Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 229–240, May 1992.

[Rao99] Satish Rao. Small distortion and volume preserving embeddings for planar and euclidean metrics. In *Proceedings of the fifteenth annual symposium on Computational geometry*, pages 300–306. ACM Press, 1999.

[RR98] Satish Rao and Andréa W. Richa. New approximation techniques for some ordering problems. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 211–218, San Francisco, California, 25–27 January 1998.

[RS95] N. Robertson and P. D. Seymour. Graph minors XIII: The disjoint paths problem. *J. Combin. Theory B*, 1995.

[RT87] Prabhakar Raghavan and Clark D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

[RU94] Prabhakar Raghavan and Eli Upfal. Efficient routing in all-optical networks. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 134–143. ACM Press, 1994.

[Sch93] Alexander Schrijver. A group-theoretical approach to disjoint paths in directed graphs. *CWI Quarterly*, 6(3):257–266, 1993.

[Sch94] Alexander Schrijver. Finding k disjoint paths in a directed planar graph. *SIAM J. Comput.*, 23:780–788, 1994.

[Sey81] P. D. Seymour. On odd cuts and planar multicommodity flows. *J. London Math. Soc.*, 42:178–192, 1981.

[Shm97] David B. Shmoys. Cut problems and their application to divide-and-conquer. In Dorit H. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 1997.

[SM90] Farhad Shahrokhi and D. W. Matula. The maximum concurrent flow problem. *Journal of the ACM (JACM)*, 37(2):318–334, 1990.

[Tar74] R. Endre Tarjan. Testing graph connectivity. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 185–193, 1974.

[Tho01] Mikkel Thorup. Compact oracles for reachability and approximate distance in planar digraphs. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science*, pages 242–251, 2001.

[Tra91] S. Tragoudas. *VLSI partitioning approximation algorithms based on multicommodity flow and other techniques*. PhD thesis, University of Texas, Dallas, 1991.

[VA99] K. R. Varadarajan and PankajK. Agarwal. Approximation algorithms for bipartite and non-bipartite matching in the plane. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 805–814, 1999.

[Vai89] P. M. Vaidya. Geometry helps in matching. *SIAM J. Comput.*, 18:1201–1225, 1989.

[Val81] L. G. Valiant. Universality consideration in vlsi circuits. *IEEE Trans. Comput.*, 30(2):135–140, 1981.

[Wei97] Karsten Weihe. Maximum (s, t)-flows in planar networks in $O(|V| \log |V|)$-time. *Journal of Computer and System Sciences*, 55(3):454–476, 1997.

[WW95] Dorothea Wagner and Karsten Weihe. A linear-time algorithm for edge-disjoint paths in planar graphs. *Combinatorica*, 15(1):135–150, 1995.